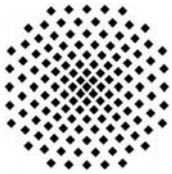

Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

Realizing Enterprise Integration Patterns in WebSphere

Thorsten Scheibler, Frank Leymann

Report 2005/09
October 20, 2005

Institut für Architektur von Anwendungssystemen

Universitätsstr. 38
70569 Stuttgart
Germany

CR: C.2.4, D.1.7, D.2.11, D.2.12

Summary

Over the last few years, patterns became focus of many activities in both, software development and research. Because of the financial significance of enterprise application integration (EAI) technologies corresponding patterns in this area are especially important and, thus, found a lot of interest. Even a standard textbook has been well-established in this space [2].

People are asking for guidelines about how to use the patterns from [2] in their environment. [1] provides a sample integration scenario together with guidelines of how to implement this integration scenario based on a subset of the patterns from [2] in the BizTalk Server 2004 environment. In this document, we use the same scenario and the same patterns as in [1] and show how to implement them in WebSphere.

Contents

1. Introduction

In [1] the authors have shown how a simple integration application designed with Enterprise Integration Patterns [2] can be implemented with the help of Microsoft BizTalk Server 2004. We now want to show how you realize the corresponding Enterprise Integration Patterns architecture with IBM WebSphere, especially exploiting WebSphere Studio Application Developer Integration Edition 5.1.1[1] (WSAD-IE). The solution will be based on *BPEL* [5] and IBM extensions of the BPEL called *BPEL+* [3]. The corresponding processes will be executed in a product called *WebSphere Process Choreographer* [4]. This runtime is included in WebSphere Application Integration Server 5.1 and is referred to as *BPC* [3].

1.1. Prerequisite

In chapter 4 "Implementing the Application" we are using WSAD-IE to implement our proposed solution. If you want to try this by your own you can take this Technical Report as a tutorial about how you can achieve that. The only prerequisite is that you are familiar with WSAD-IE: we don't go into all details and may even skip some steps which are needed for getting a running implementation. Furthermore you have to understand the basics of the Web Service Architecture [6] in particular how BPEL works together with the various Web Service technologies.

2. Overview

First, we take look at the example. A customer provides some personal information and the terms to get a loan. The loan broker needs this data to determine the best loan offer the customer is asking for. This input is sent to the loan broker. In return the customer receives the best interest rate the broker could obtain from the banks. In order to get this result, the loan broker interacts with a credit bureau to receive the credit worthiness of the customer after receiving a customer's request. This result together with the terms of the favored loan will be transferred to several loaners (i.e. banks). If these banks offer a loan based on the given input, these offers will be immediately sent back to the loan broker. The broker chooses the best choice out of the multiple quotes and returns the result to the customer. Figure 2.1.1 shows the overview of the scenario.

2.1. Requirements

The loan broker needs the following input provided by the customer:

- The customers social security number to uniquely identify him or her
- The desired loan amount in Euro
- The desired loan term in months

The customer will get following information as result:

- The interest rate of the best quote chosen by the loan broker
- A unique Quote ID from the loaner for future references

[1] http://www-306.ibm.com/software/integration/wsadie/support/

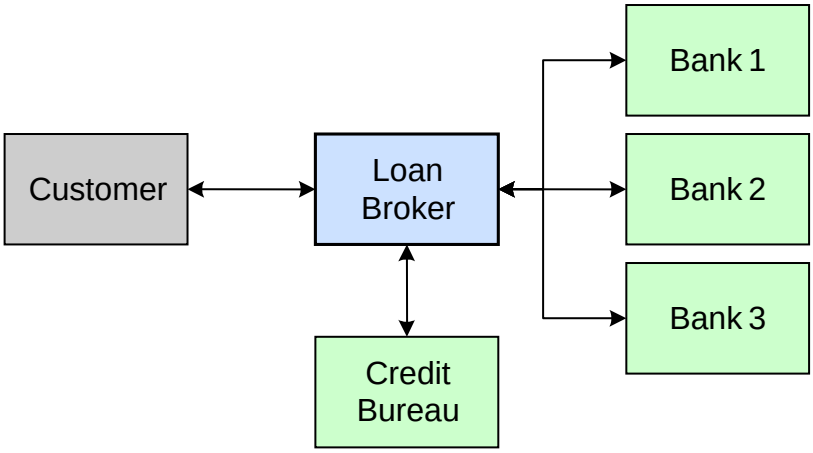

Figure 2.1.1: Overview of the example scenario

The loan broker has to interact with a credit bureau and different banks to achieve the result.

To communicate with the banks the loan broker requires information about the credit standing of the customer so he can enrich the information passed to the banks with needed pieces. The loan broker transfers the social security number to the credit bureau so that the credit bureau can uniquely identify the customer. In return the credit bureau provides the credit score and the credit history length of the costumer to the loan broker. With these information and the initial input values the broker can contact different banks to get different loan quotes.

We define the communication with three banks. Each of them offers specialized loans for different kinds of costumer. The loan broker will filter request to banks so that they are not burdened with quote requests from customers which don't fit to their customer portfolio. Each bank defines its customer portfolio with preferences describing the wanted duration of the customer's credit history and the credit score rating. The preferences for the three banks look like this:

Bank 1 – Customer Bank for the widest range of customers:
Credit Score has to have 500 or higher and the credit history has to be at least 5 months.

Bank 2 – Exclusive Bank for the best rates offered to a top-end clientele:
Credit Score has to have 700 and higher and the credit history length has to be at least 10 months.

Bank 3 – Loan Shark which gives loans to everybody and makes no restrictions to credit score neither to credit history length.

The banks will decide individually whether they will expose an offer or not. In both cases the bank has to return a reply to the loan broker. If it won't make an offer this conclusion has to be included in the reply. The loan broker can determine the result afterwards. With this procedure the loan broker can distinguish a missing response from the decision of not providing an offer.

2.2. Designing with Patterns

To create an architecture for the loan broker application we first have to determine the processing steps for getting the appropriate information for the customer. The loan broker application has to do the following:

1. Receive requests from the customer.

2. Obtain credit score from the credit bureau for every customer.
3. Select the banks to be contacted.
4. Request every selected bank for loan quotes.
5. Receive requests from each bank.
6. Select the best offer out of the requests.
7. Reply the best offer back to the customer.

Each single step can be modeled with the help of one or more patterns. In each step we may choose different kinds of patterns. We simplify the procedure of selecting a specific pattern by leaving out the discussions of the benefits and drawbacks of each solution. We will just explain why the chosen pattern is right for our architecture. Furthermore we won't explain in detail what the pattern is about. If you're interested in more detailed information see [2].

Let's begin with the first step where a customer sends a request to the loan broker. The customer only sees an interface which offers a special operation or service to him. The possibly complex structure of the underlying system is hidden. These considerations lead to a pattern which is called *"Service Interface"*. While implementing this pattern you have to be aware of some design decisions, e.g. you have to think about the communication style. In our case we use synchronous communication between the customer and the loan broker application.

Before a bank can be contacted more information is needed. Because of that we have to enrich the content of the customer's request of the first step with additional data. As the description already suggests we use the *"Content Enricher"* patter for that purpose. In our case the content enricher communicates with an external credit bureau to get the required information.

After enriching the request in step 2 we have to walk through the next steps and have to send it to multiple banks. For that we have to define a channel for each connected bank. Depending on the incoming message the list of recipients is determined, and the message is forwarded to all channels associated with the recipients in the list. Afterwards we have to collect all the response messages of the different banks and form a single message out of them. That behavior can be realized based on the *"Scatter-Gather"* pattern which broadcasts a message to multiple recipients and aggregates the different responses into a single message. In our case the "inner structure" of the scatter-gather pattern consists of a *"Recipient List"* which contains a rule base for selecting the appropriate recipients and an *"Aggregator"* which selects the best quote of the banks and forms a single message out of the result. The aggregator service may be used by multiple processes at a time. Because of that we need a *"Correlation Identifier"* which is inserted into the response messages of the banks.

With the *"Scatter-Gather"* Pattern we have processed steps 3 to 6. To finalize our loan broker example process the result has to be sent back to the calling customer. This is done by the already selected *"Service Interface"* pattern where the customer calls a method to get a response. But before we can send back the response message we have to transform the result message of the aggregator into a format the customer expects. We utilize the *"Message Translator"* pattern which translates one data format into another to accomplish that.

Put all these patterns together results in the architecture of our overall solution shown in Figure 2.2.1.

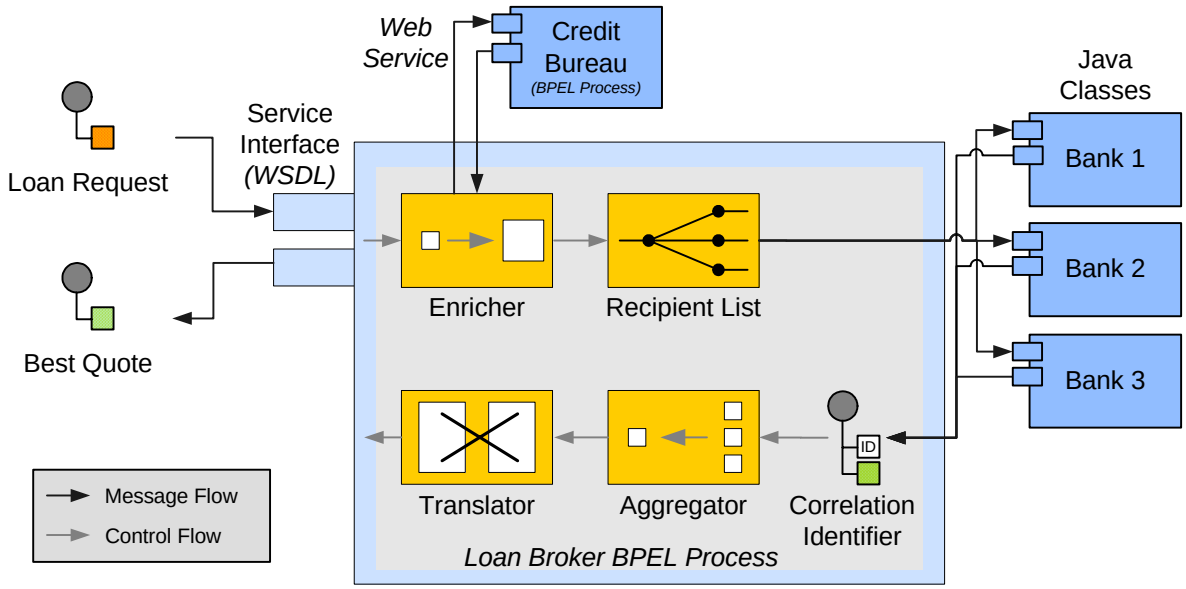

Figure 2.2.1: The architecture with involved patterns

3. Mapping Patterns to WSAD-IE

After choosing our patterns based on which we described the architecture of our solution we have to consider how we can implement these patterns within a J2EE environment offered by the WSAD-IE including Web Service capabilities and BPEL execution support.

Service Interface

A *"Service Interface"* within a Web Service environment is simply represented by a port type within *Web Service Description Language (WSDL)* file. Inside this file all information about the port type, its bindings and ports is specified. Because we want to build a business process to calculate the loan broker outcome we need a BPEL based business process in conjunction with the WSDL file. Within this process the other patterns will be implemented. The WSDL file will also be used to define the *Partner Link Type* of the *Partner Role* of the business process which can be used by the customer's application.

Content Enricher

The content enricher consists of two parts which both must be implemented. The first part to implement is the credit bureau service itself which is used to enrich the customer's request with additional information. Because we want to show the versatility of WSAD-IE we use different methods of implementing some features of the loan broker scenario. For the credit bureau we choose to calculate the additional data within a separate business process which will be encapsulated as Web Service.

The second part of the *"Content Enricher"* is implemented within the loan broker business process. This part will handle two steps. First we have to call the credit bureau and afterwards we have to add additional data to the customer's request. These two steps will be implemented by an *"Invoke"* activity followed by a *"Transform"* activity.

Recipient List

The recipient list pattern gets a single message as input, copies it and transfers it to multiple recipients. So we can process the recipient list within three steps. In the first step the recipients of an appropriate message have to be determined. In the second step the message has to be copied. This can be done in two ways: first, copy the incoming message before selecting the recipients; second, copy the messages after determining the recipients. We decided for the first variant because we want the copy activity at one single place at the price of a certain overhead because we copy messages that sometimes won't be used afterwards. But since we're just talking to three banks that overhead is affordable. The last step is to send the copied messages to the previously selected banks. In our case this is done in parallel.

To implement this behavior we're using a *"Flow"* activity wherein three flows are modeled, one for each bank. To select the recipients we include *"Switch"* activities followed by a *"Case"* activity, and for copying messages we utilize an *"Assign"* activity. Sending requests to the banks is done by *"Invoke"* activities.

Correlation Identifier

A correlation identifier pattern (note, that this is different from BPEL's build-in feature of correlation sets) simply takes an identifier and adds it to a message. So first we have to generate a unique identifier for our process. Afterwards we just construct a new message with the help of an *"Assign"* activity- one for each bank response. These messages can be sent to the according aggregator.

Aggregator

The aggregator is collecting all bank responses and calculates the best result out of the incoming messages. We model this with the help of a *"Flow"* activity with three different flows for each possible response message. In the flow the aggregation service is called with an *"Invoke"* activity and the response message is added to the response set included within the aggregator. Afterwards the service is called one more time to request the calculation of the best result.

The aggregator will be implemented as Java singleton [7] so that we can be sure that all messages are added to the same instance of the service implementation. Because of that we have to be aware of correlation so that the aggregation service knows which incoming message belongs to which caller (i.e. process). Furthermore the service needs a correlation ID to identify which messages belong together when calculating the best result. That's very important when the aggregation service is called by multiple clients or multiple times during its lifetime.

Message Translator

The message translator can be easily implemented by a special activity within WSAD-IE: With the help of the *"Transform"* activity it's possible to transform one or more incoming messages to one outgoing message.

4. Implementing the Application

4.1. The Credit Bureau Business Process

The credit bureau is required to enrich the requests for the banks with additional information about the customer's credit status. The bureau provides this service as a Web Service. The message types, port types and bindings are described in a WSDL-file. A BPEL-process which will determine the needed information resides behind the Web Service. The process acquires the credit worthiness of the customer with a numerical credit score. This score together with the duration of the credit history will be used by the loaners to determine the risk of lending money to the customer. For the sake of simplicity we will use randomly generated values instead of a complex mathematical calculation algorithm to get the values.

The loan broker utilizes the credit bureau by calling the Web Service passing the social security number (SSN) as parameter which will uniquely identify the customer. As response the broker expects a credit score as well as the length of the credit history.

Implementation

We have to walk through a number of steps to construct the credit bureau Web Service (i.e. the business process of the credit bureau):

1. Define the message formats of all incoming and outgoing messages.
2. Define the operations of the Web Service.
3. Create a new business process for the credit bureau.
4. Define the partner link type and partner role of the created BPEL-process.
5. Define the variables inside the BPEL-process which will represent the incoming and outgoing messages.
6. Add a receive activity to the process to receive incoming messages.
7. Construct a reply message with the help of a so called "*Java Snippet*".
8. Add a reply shape to the process in order to send the result of the calculation.
9. Deploy the process to the WebSphere Application Server.
10. Publish the generated application onto the application server.
11. Start the application on the server.
12. Test the application with the help of the internal test client.

Step 1 – The message formats

Before we start with the design of the business process we first need to specify the incoming and outgoing messages of the process. The format will be designed with the help of XML Schema. WSAD-IE provides us with the possibility of creating XSD files within a graphical XML editor.

When using this XML schema inside a WSDL file which represents the credit bureau process, WSAD-IE will generate corresponding messages as Java classes (see Step 3 for details).

The schema which represents request and response messages should look like Figure 4.1.1 in WSAD-IE XML Schema designer.

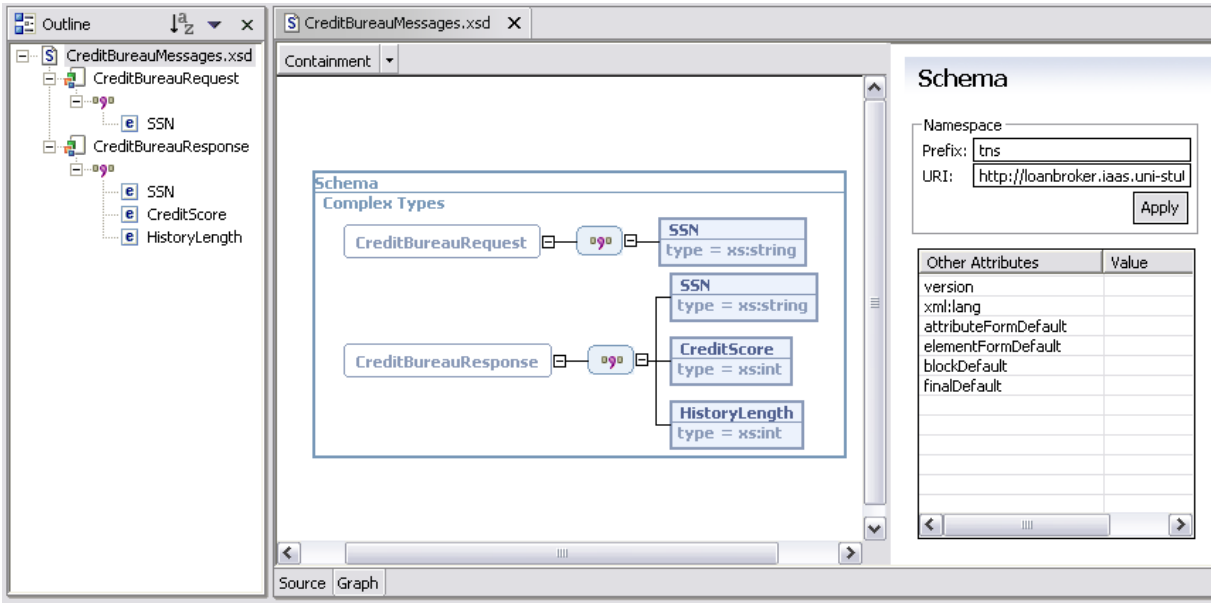

Figure 4.1.1: Credit Bureau Messages Schema

Step 2 – Define the operations

The next step in creating the credit bureau process is to define the operations and the input and output parameters, i.e. the messages. Because we want to publish that process as a Web Service we will be designing a WSDL-file to describe the behavior of the service. At this part we don't need to take care of the binding. We only need the description of the port types together with including operations and messages. To design such a file WSAD-IE provides a WSDL editor.

For our purpose we need the port type "CreditBureauPort" which offers only one operation which is called "GetCreditScore". The operation has an input and an output message. Each message part has the format of the previous defined schemas (see Figure 4.1.2).

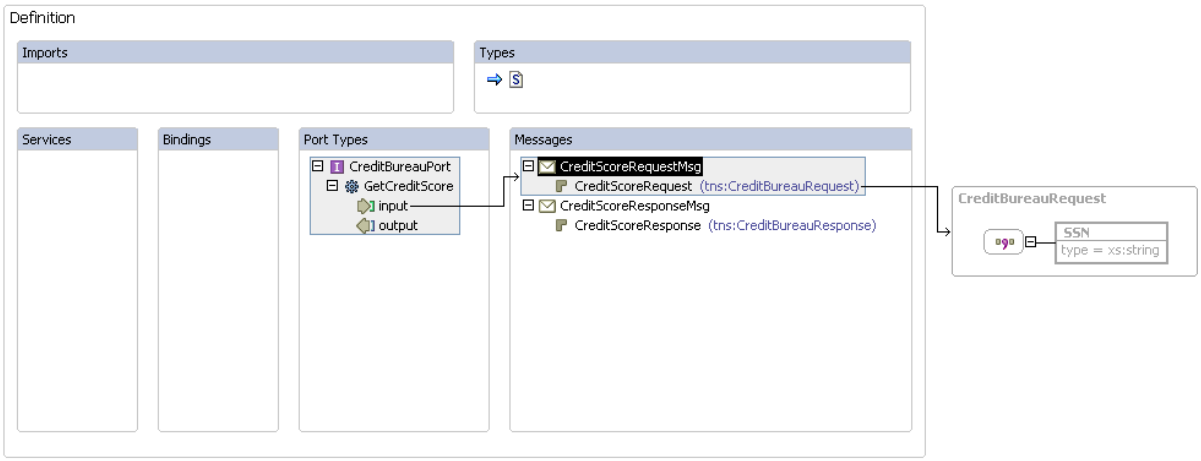

Figure 4.1.2: Credit Bureau Port Type definition

Step 3 – Create a new business process

The next step is the creation of a business process. We have to create a new empty business process so we can customize it for our needs. WSAD-IE is working with BPEL to describe and execute a business process. After creating the

process we have to delete the default values of partner link, variables and receive and reply shapes at first. If the process and its corresponding WSDL description are cleaned we can go on to the next step.

Step 4 – Define the partner link role

Each BPEL process must have a partner role. The partner role is described within the partner link tag which itself can be described by a WSDL file. We use the description created in step 2. Import the file into the business process and change the implementation as shown in the picture (see Figure 4.1.3).
Now we have created a business process which will take over the partner link role as "CreditBureau" within the partner link type "CreditBureaoPortLT" which offers the port type "CreditBureauPort" with the operation "GetCreditScore" to its possible users.

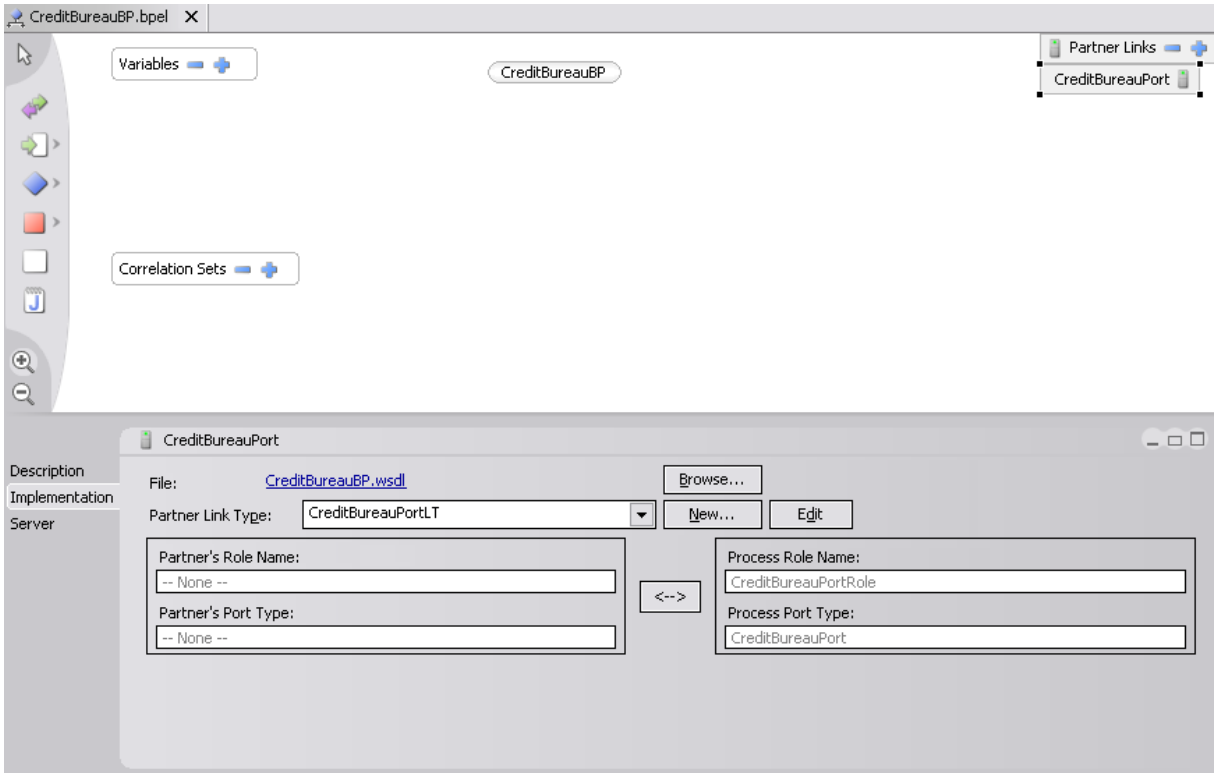

Figure 4.1.3: Implementation of the partner link

Step 5 – Define the variables inside the process

After the declaration of the partner link we have to create the variables which represent the incoming (request) and outgoing (response) messages during the existence of a process instance. Add a variable to the business process by pressing the "add" button beside the variables label. Enter "CreditScoreRequest" as the variable representing the request sent to the process (or better Web Service) and "CreditScoreResponse" as the variable representing the response. As types we have to refer to the previously (step 2) created WSDL file and select the according message. In case of "CreditScoreRequest" you have to choose "CreditScoreRequestMsg" as type. During the creation of the variables WSAD-IE generates Java Bean classes. In our case two Java classes are generated namely "CreditBureauRequest.java" and "CreditBureaResponse.java".

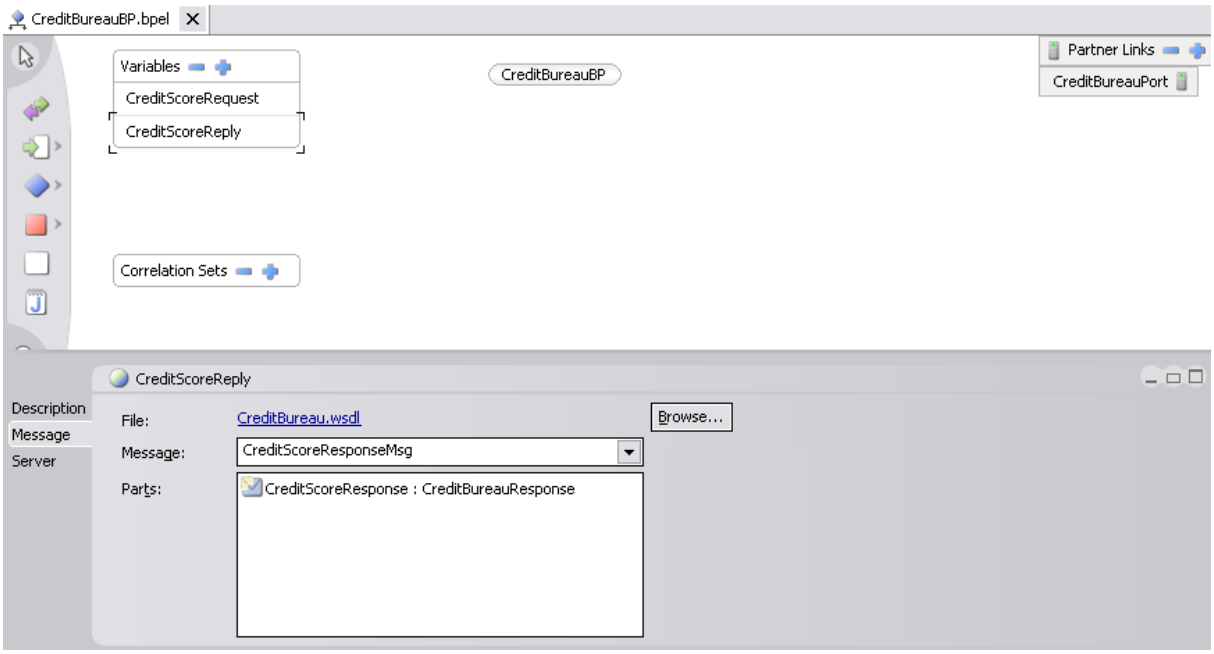

Figure 4.1.4: Declared variables in the business process

Step 6 – Add a receive activity

The process is now prepared for adding a receive shape which will accept incoming requests on the port type "CreditBureauPort" and the operation "GetCreditScore". After adding the shape we have to connect it to the corresponding partner link select the operation and assign the incoming message variable. By now the process is able to receive an incoming "CreditScoreRequestMessage" message on the appropriate port and can access therein included values.

Step 7 – Construct the reply message

In this step the real work of the process has to be realized. In our case the computation and determination of the credit score will be a single and very simple step. In real life this work may consist of different steps, maybe contact multiple backend systems and do a complicated computation. For the sake of simplicity our calculation of the credit score depends on some random values.

To construct the response message we have to add a "*Java Snippet*" activity to the process and fill it with simple Java Code shown in Listing 4.1.1.

```
java.util.Random random = new java.util.Random();
int creditScore = random.nextInt(600) + 300;
int historyLength = random.nextInt(19) + 1;

CreditBureauResponse resp = new CreditBureauResponse();
resp.setHistoryLength(historyLength);
resp.setCreditScore(creditScore);
resp.setSSN(getCreditBureauRequest().getCreditScoreRequest().getSSN());

CreditScoreResponseMsgMessage msg = new CreditScoreResponseMsgMessage();
msg.setCreditScoreResponse(resp);
setCreditBureauResponse(msg);
```

Listing 4.1.1: Construct the credit bureau reply

The code creates two random values, one for the credit score and one for the history length. Afterwards a new message is created and filled with the generated values. This message will be put into the context of the BPEL process so that subsequent activities can work with the variable. In our case only the reply activity needs the variable to send the response back to the requesting consumer.

Step 8 – Add a reply activity

To finish the process we have to add a reply shape which takes a variable out of the process context and sends it to the requesting customer. At first we have to set the partner link of the reply activity, select the operation and the variable that contains the outgoing message.

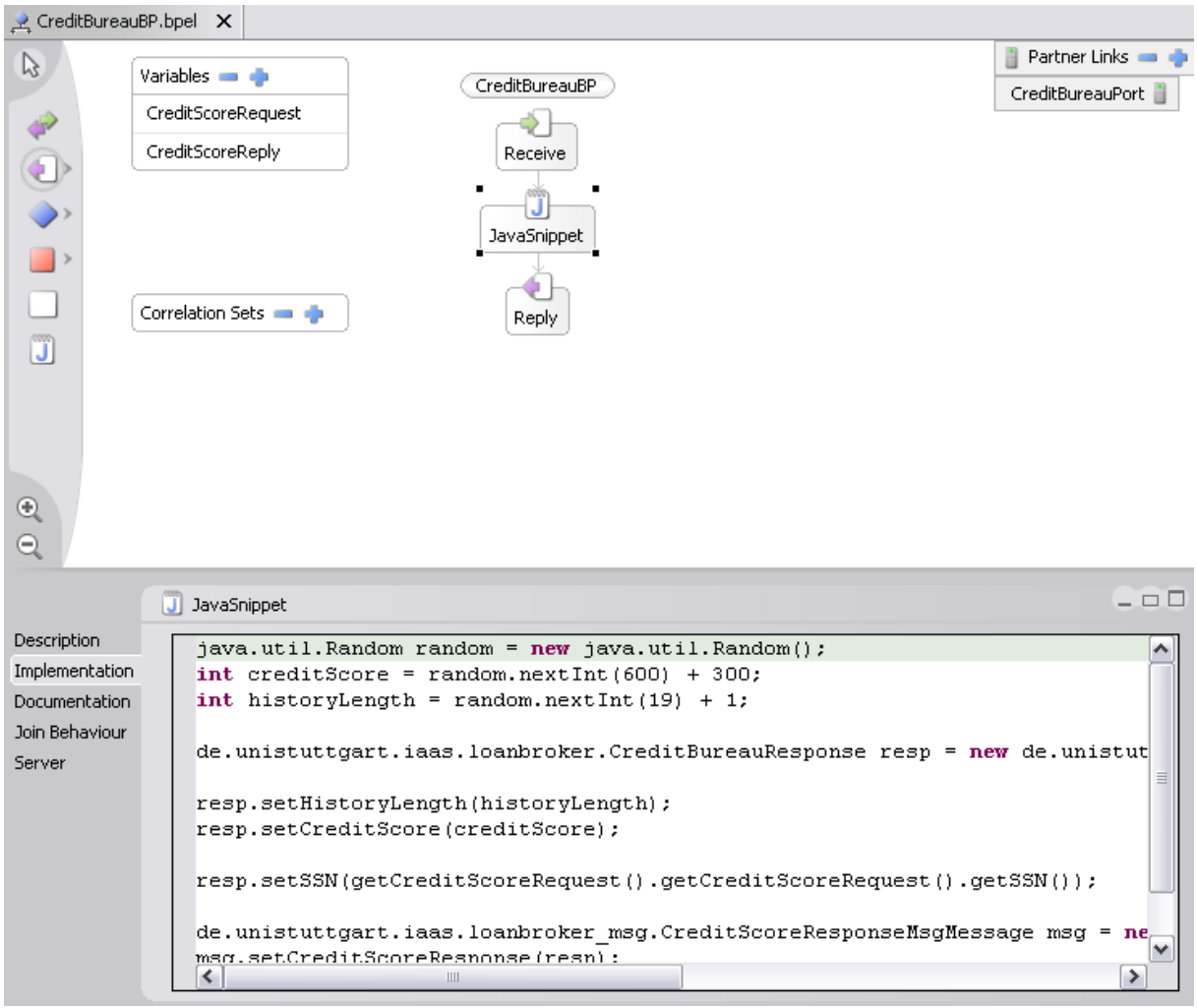

Figure 4.1.5: The overall credit bureau business process

The credit bureau business process is now ready for computing the credit score for a customer.

Step 9 – Deploy the process

After creating the business process it has to be made available on an application server. In our case we're using an integrated WebSphere Application Integration Server (WAS) Version 5.1. First set up such a server. This procedure is far

beyond our scope so read continuative material which you can find at the help section of WSAD-IE. Afterwards we have to deploy our business process. WSAD-IE provides us with functionality of generating deployment code. We only have to adjust the bindings that shall be generated for the port type of the business process. Because the process, i.e. the Web Service, is called by an external partner (in our case the loan broker process) the generator is set to generate the binding for SOAP over HTTP. Other bindings aren't needed at present. Now WSAD-IE generates a lot of projects, the WSDL files, EJBs, deployment descriptors and so on. In short, everything needed for the publishing of the process on a WAS.

Step 10 – Publish the generated application

Publish the application containing the process on the server after deployment of the credit bureau process. Add the previously generated EAR project to the server's application projects. Afterwards publish the application. This is done by selecting the option in the context menu entry of the server. The job is done automatically and the database is being updated so that no further interaction is needed.

Step 11 – Start the application

After publishing the application it's required to start the application in order to use it. This is done by selecting the appropriate menu item out of the context menu entry of the server. The Server tries to start the application. The result is shown at the server's console.

Step 12 – Test the application

After publishing and starting the application on the server we can test the business process with the help of WSAD-IE. The integrated *"Business Process Web Client"* offers the possibility among other things to test business processes. This is done by entering the values of the request message into a web front-end. Afterwards the web client shows the results, i.e. the values of the response message, generated by the business process. The following two screenshots show this proceeding.

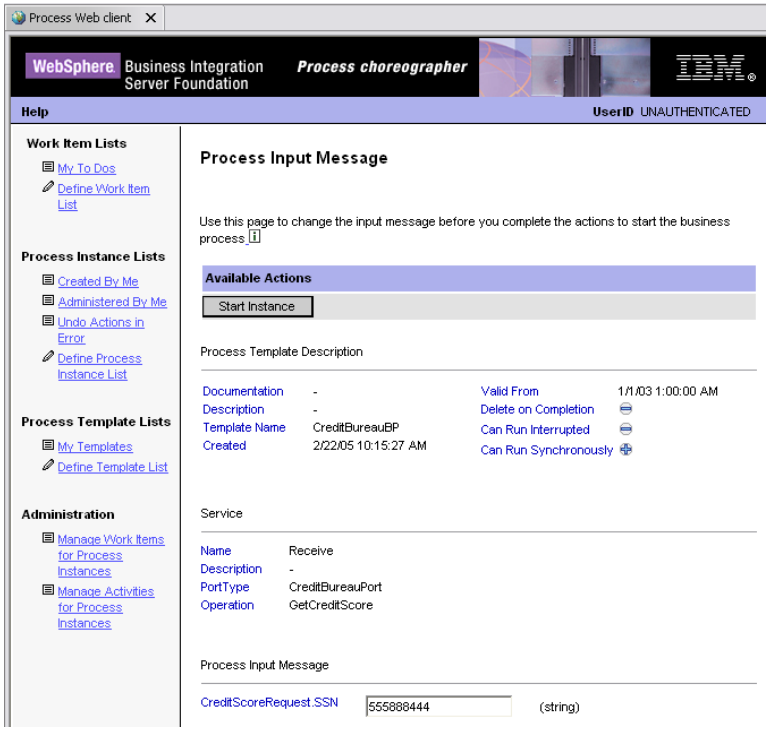

Figure 4.1.6: Enter request values for a business process

As reminder the credit bureau web service requires a SSN as input value and returns the same SSN together with a credit score and a history length. In our example we set the SSN to '555888444' and send this request (encapsulated in a request message) to the web service. As result we get the same SSN along with random values for credit score, '860', and for the history length, '7'.

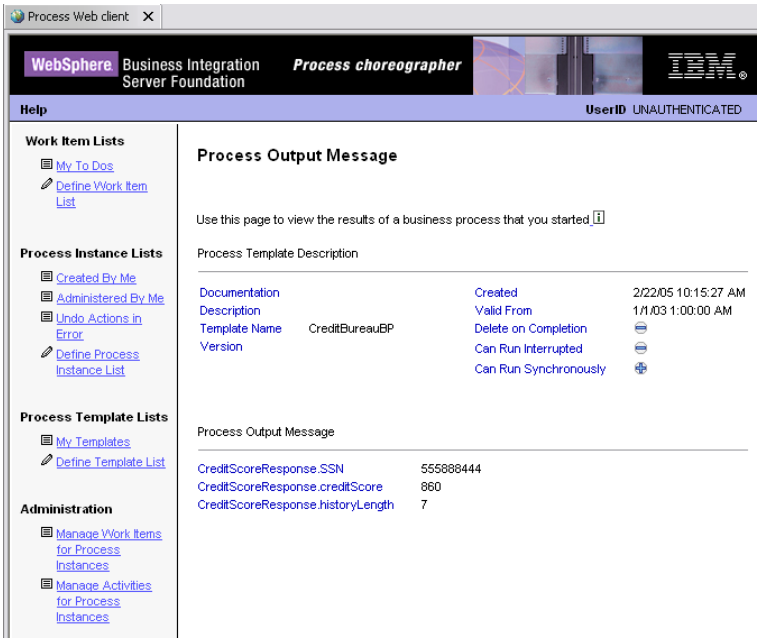

Figure 4.1.7: Overview of the result of the business process

4.2. The Bank Web Services

After getting the result of the credit bureau, the loan broker needs to interact with different banks to obtain loan offers. Comparable to the credit bureau the banks are Web Services which generate random values as reply to a request.

The banks will be designed as Java classes which will expose a method to the caller that allows passing a quote request message. This method will reply a quote reply message. All three banks are very similar so that we will implement an abstract bank class which will encapsulate the calculation of the interest rate. The three bank instances- named "CreditBank1", "CreditBank2" and CreditBank3"- will inherit form the abstract class "CreditBankAbstr" and implement the bank specific values like name, maximum loan term and rate premium.

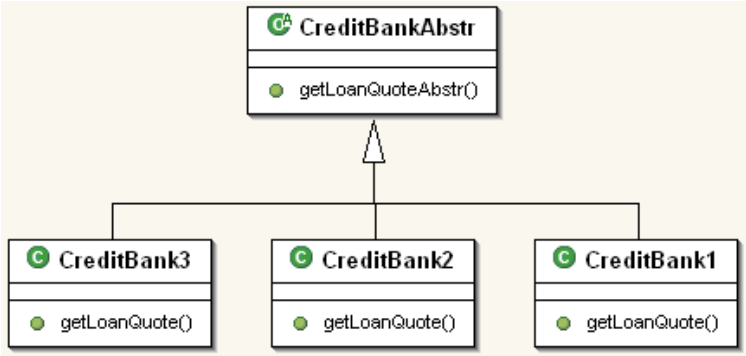

Figure 4.2.1: Overview of the banks architecture

We want to use three different kinds of banks. Each of them is distinguished by different bank names and a different customer's target group. The three banks differ from each other by the minimum premium rate, which determines their profit margin, and the maximum loan term the bank is willing to accept.
Our example offers these three banks:

Bank 1 – *Customer Bank* services a wide range of customers. It charges a rate premium at 2.0% and willing to engage a maximum loan term of 48 months.

Bank 2 – *Exclusive Bank* offers the best loans but only to a top-end clientele. The bank only charges a 1.8% rate premium and is willing to accept a maximum loan term of 60 months.

Bank 3 – *Loan Shark* offers quotes to all interested customers regardless of the credit score and credit history length. Because of that it charges a high premium rate at 4.0% and a maximum loan term of 72 months.

Implementation

The following steps are involved while creating the Web Services of the multiple banks:

1. Define the message structures of the incoming and outgoing messages
2. Generate the Java classes which represents this message types
3. Implement the abstract Bank class with the quote calculation algorithm
4. Implement three different instances of the banks
5. Generate the Web Services for each of the bank
6. Test the Web Services with the help of a service proxy

Step 1 – Define message structures

At first we have to specify the message structures that will be used by the credit bank Web Services. We also use XML Schema at this point to design the messages. The proceeding will be the same as in the previous chapter.

With the help of the integrated XML Schema Editor of WSAD-IE 5.1 we create the schema shown in Figure 4.2.2 for the incoming and outgoing messages of the credit bank Web Service.

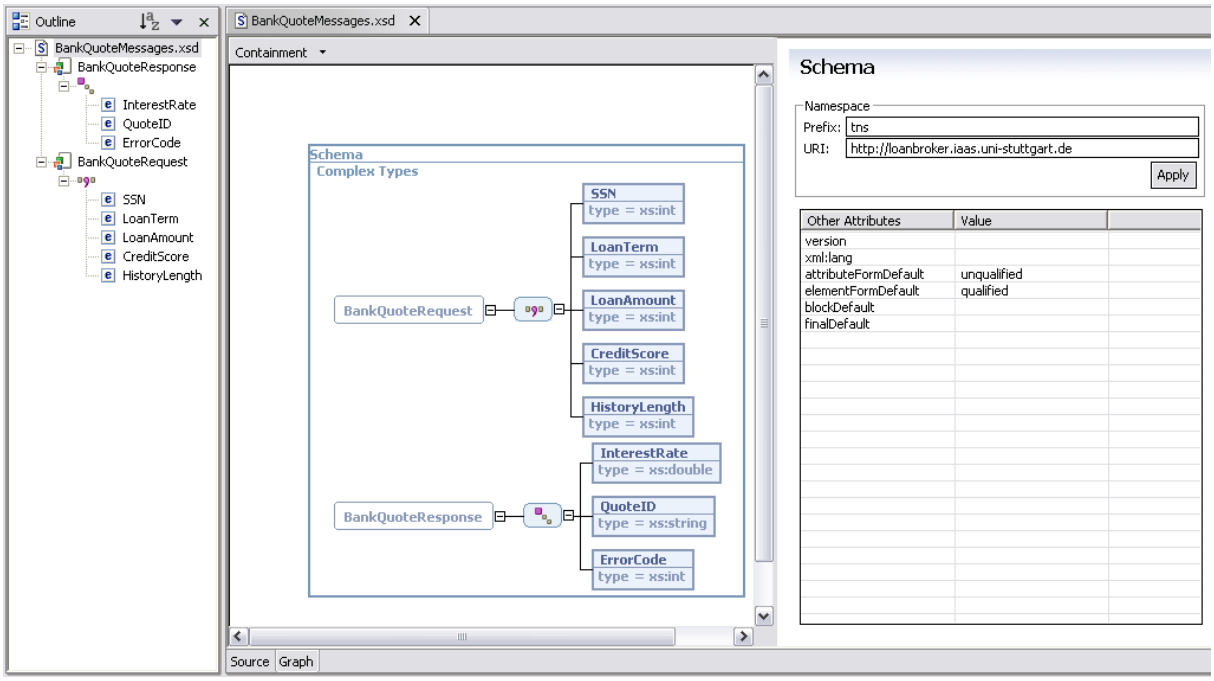

Figure 4.2.2: Credit Bank Messages Structures

Step 2 – Generate Java message classes

After designing the message formats and before implementing the Java classes we first have to create the Java files which represents the two different messages. WSAD-IE 5.1 provides us with functionality to generate Java classes out of a XML schema file. In our case the tool will generate two Java classes named "BankQuoteRequest.java" and "BankQuoteResponse.java". With these two classes we can specify the signature of the method which each instance of the credit banks has to implement.

Step 3 – Implement the abstract Bank class

Before implementing the concrete instances of the banks the abstract class with the main logic has to be implemented. This class provides a method which separates the computation of the interest rate from the real instances. With this circumstance the instances of the banks do not have to deal with the computation. This is possible because the calculation of the interest rate only differs in parameters which will be set in each of the instances. The implementation of the abstract class is shown in Listing 4.2.1.

```
public abstract class CreditBankAbstr {

   public double PRIME_RATE = 2.0;
```

```
   private String bankName;
   private double ratePremium;
   private int maxLoanTerm;

   /** The constructor */
   public CreditBankAbstr() {
      bankName = getBankName();
      ratePremium = getRatePremium();
      maxLoanTerm = getMaxLoanTerm();
   }

   protected BankQuoteReply getLoanQuoteAbstr(BankQuoteRequest request) {

      BankQuoteReply reply = new BankQuoteReply();

      if (request.getLoanTerm() <= maxLoanTerm) {
         reply.setInterestRate(
            this.PRIME_RATE + ratePremium + (request.getLoanTerm() / 12) /
             10 + (new Random()).nextInt(10) / 10);
         reply.setErrorCode(new Integer(0));
      } else {
         reply.setInterestRate(0.0);
         reply.setErrorCode(new Integer(1));
      }
      reply.setQuoteID(bankName + (new Random()).nextInt(100000) + 99999);
      return reply;
   }

   /** To be implemented by inheriting classes. */
   public abstract BankQuoteReply getLoanQuote(
      BankQuoteRequest argBankQuoteRequest);

   /** To be implemented by inheriting classes. */
   protected abstract String getBankName();

   /** To be implemented by inheriting classes. */
   protected abstract double getRatePremium();

   /** To be implemented by inheriting classes. */
   protected abstract int getMaxLoanTerm();
}
```

Listing 4.2.1: The abstract bank class

As you can see, the whole "magic" happens inside the method 'getLoanQuoteAbstr' where the answer of the Web Service is being created with random values. In real life this computation is much more complex and would involve backend and information systems. In our case this calculation fits our needs.

The information of not offering a loan quote is being handled in the else branch where the interest rate is set to 0.0 and the error code is set to 1. With this information the caller (in our case the loan broker) can distinguish if a bank is willing to offer a loan or not.

Step 4 – Implement three instances of the banks

The next step is to implement three different instances of the abstract credit bank class. The instances inherit from the abstract class and have to implement four different methods. With the help of these methods each bank differs from the

others by setting bank specific values. As mentioned in 2.1 “Requirements” the differences are shown in the different code pieces. The source code of CreditBank1.java is shown in Listing 4.2.2.

```
...
public BankQuoteReply getLoanQuote(BankQuoteRequest argBankQuoteRequest) {
   return getLoanQuoteAbstr(argBankQuoteRequest);
}

protected String getBankName() {
   return "ConsumerBank";
}

protected double getRatePremium() {
   return 2.0;
}

protected int getMaxLoanTerm() {
   return 48;
}
...
```

Listing 4.2.2: CreditBank1 – Customer Bank

CreditBank2.java implements the abstract methods as shown in Listing 4.2.3.

```
...
public BankQuoteReply getLoanQuote(BankQuoteRequest argBankQuoteRequest) {
   return  getLoanQuoteAbstr(argBankQuoteRequest);
}

protected String getBankName() {
   return "ExclusiveBank";
}

protected double getRatePremium() {
   return 1.8;
}

protected int getMaxLoanTerm() {
   return 60;
}
...
```

Listing 4.2.3: CreditBank2 – Exclusive Bank

At last have a look at the code of CreditBank3.java in Listing 4.2.4.

```
...
public BankQuoteReply getLoanQuote(BankQuoteRequest argBankQuoteRequest) {
   return  getLoanQuoteAbstr(argBankQuoteRequest);
}

protected String getBankName() {
   return "Loan Shark";
}

protected double getRatePremium() {
   return 4.0;
}

protected int getMaxLoanTerm() {
```

```
    return 72;
}
...
```

Listing 4.2.4: CreditBank3 – Loan Shark

Step 5 – Generate the Web Services

After implementing three instances of the banks we have to wrap these implementations inside a Web Service. So we have to create a description of the Web Service and a binding to the Java implementation. WSAD-IE 5.1 provides us with functionality of generating a Web Service out of a Java class.

We have to walk through the procedure three times for every single credit bank. For simplicity we will only present the result for the Customer Bank (see Figure 4.2.3). The other two Web Services are created analogues.

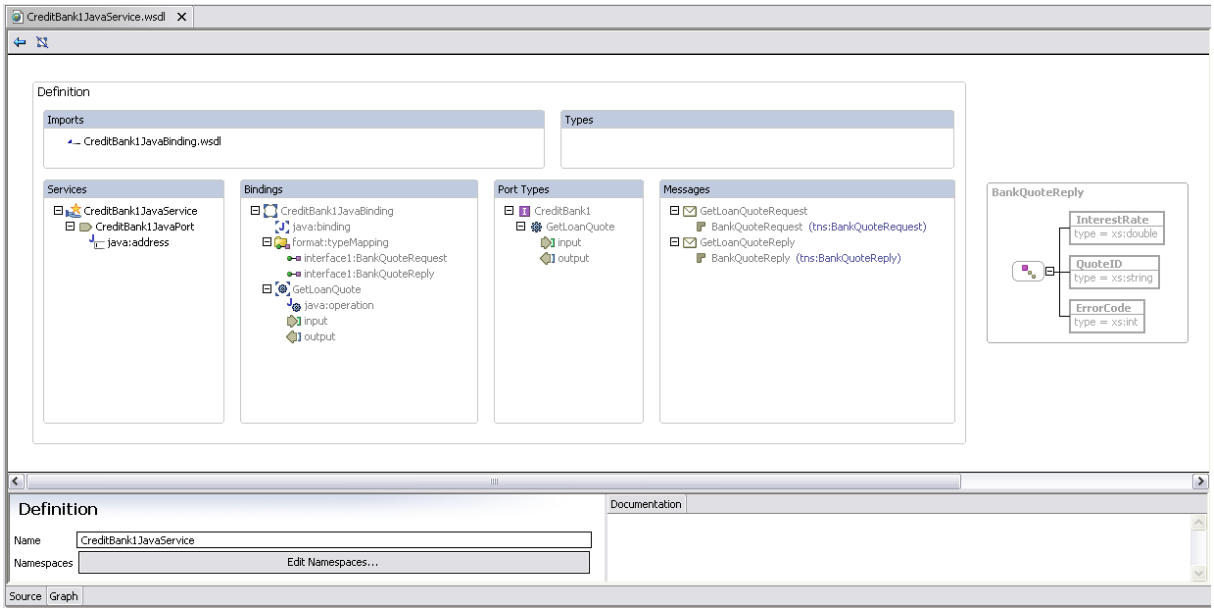

Figure 4.2.3: CreditBank1.wsdl

Step 6 – Test the Web Services

To test the Credit Bank Web Services we can use a tool of WSAD-IE. This tool generates a so called Service Proxy out of a WSDL file. With the help of this proxy you can test a Web Service without the need of something else than this class. Simply generate the proxy class an implement a test case as show in Listing 4.2.5.

```
...
/**
 * main method (for proxy unit testing)
 * @generated
 */
public static void main(String[] args) {

try {
  CreditBank1Proxy aProxy = new CreditBank1Proxy();
  // user code begin {proxy_method_calls}
  BankQuoteRequest req = new BankQuoteRequest();
  req.setCreditScore(250);
  req.setHistoryLength(60);
```

```
  req.setLoanAmount(45000);
  req.setLoanTerm(45);
  req.setSSN(47110815);
  BankQuoteResponse resp  = aProxy.getLoanQuote(req);
  System.out.println("Bank Quote Response of CreditBank1 succesfull.");
  System.out.println("  InterestRate: " + resp.getInterestRate() +
                     "  - ErrorCode: " + resp.getErrorCode());
  // user code end
} catch (Exception e) {
  // user code begin {exception_handling}
  e.printStackTrace();
  // user code end
}
...
```

Listing 4.2.5: Service proxy with a test case

When you run this proxy the Java Web Service will be tested. If everything goes right you see the output of the test case in the console.

4.3. The Loan Broker Business Process

Until this step we only did preparatory work so that we are able to build the main process of the example. Now we are ready to design the process and integrate our previously created Web Services into the main flow.

Implementation

In order to implement the Loan Broker process, following steps have to be performed:

1. Define message schemas for the incoming and outgoing messages.
2. Define the operations of the Web Service.
3. Create a new business process for the loan broker.
4. Define the partner link role of the created BPEL-process.
5. Define the variables inside the BPEL-process which will represent the incoming and outgoing messages.
6. Enhance the process with a receive shape to be able to receive incoming messages.
7. Insert the interaction with the Credit Bureau Web Service.
8. Construct the interaction with the banks.
9. Build the aggregator so that the different replies of the banks will be aggregated into one single message.
10. Construct the reply message of the loan broker business process.
11. In order to send the result of the calculation, add a reply shape to the process.
12. Deploy the process to the WebSphere Application Server.
13. Publish the generated application onto the application server.
14. Start the application on the server.
15. Test the application with the help of the internal test client.

Step 1 – The message schemas

The starting point in the creation of the loan broker process is the same as in the sections above. At first we need to define the incoming and outgoing message

formats. As we're using the same technology as mentioned earlier we don't need to go through it anymore.

The message schemas should look like shown in Figure 4.3.1.

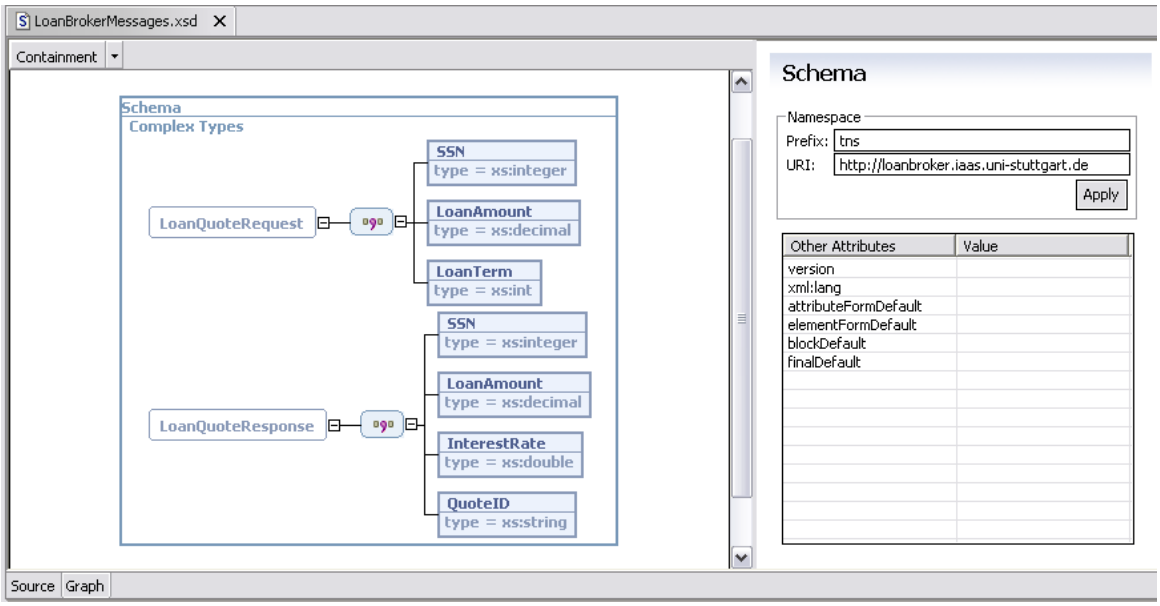

Figure 4.3.1: Loan broker message schemas

Step 2 – Define the operations

The next step has to deal with the operations that our loan broker offers to its clients. As we're creating a Web Service we have to describe the operations in a WSDL file. The graphical representation of the file is shown in Figure 4.3.2.

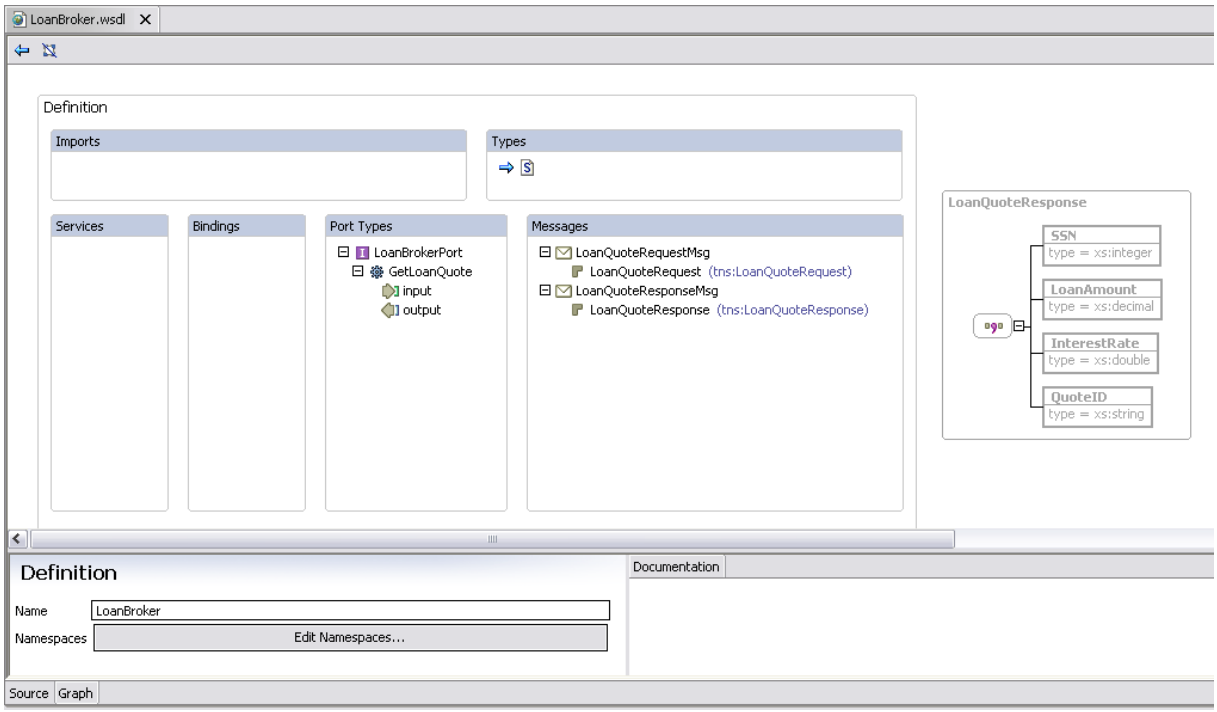

Figure 4.3.2: Loan broker WSDL

The loan broker only has to offer one operation. The client gets its loan quote by calling the "`GetLoanQuote`" operation with the message in the format defined in the first step. As result he gets the best loan quote the loan broker could obtain for him.

Step 3 – Create a new business process

Now we need a new empty business process. See the appropriate step at the credit bureau process creation in section 4.1 "The Credit Bureau Business Process". Afterwards we get a cleaned BPEL and a cleaned WSDL file and go on to the next step.

Step 4 – Define the partner link role

Since the process, i.e. Web Service, will be used by other partners, we have to specify the role of our business process. Import the WSDL file which was created two steps earlier and set the partner link like it is shown in Figure 4.3.3.

If you did so the loan broker process will take over the role "LoanBrokerPortRole" at the port "LoanBrokerPort". Thus it can accept messages which where send to this port, process the message and send a response back to the caller.

Step 5 – Define the request/response variables inside the process

Accordingly to the credit bureau process the next step is to create the variables for the request and response messages inside the BPEL process (see Figure 4.3.4). For the incoming "LoanQuoteRequest" the message type has to be set to the type specified a few steps earlier in the "LoanBroker.wsdl" file. The same procedure has to be done with the "LoanQuoteResponse" variable. During this creation WSAD-IE generates Java files for each of the message objects. These files are needed during the next steps of the loan broker business process.

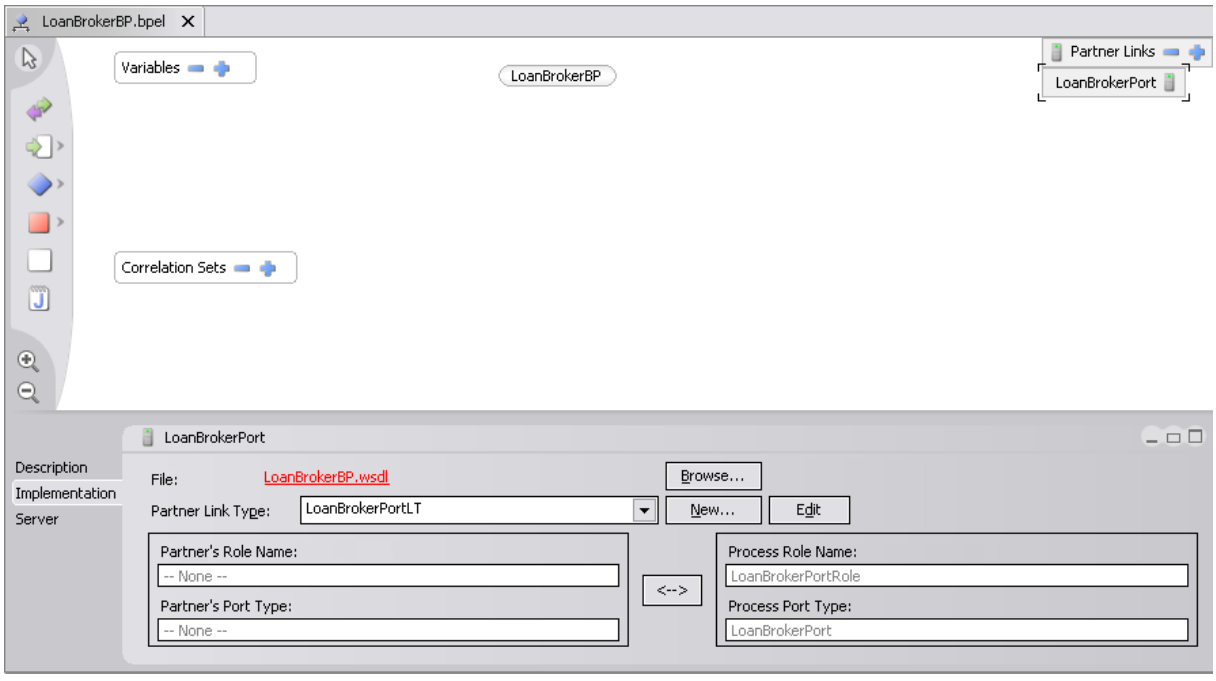

Figure 4.3.3: Implementation of the partner link

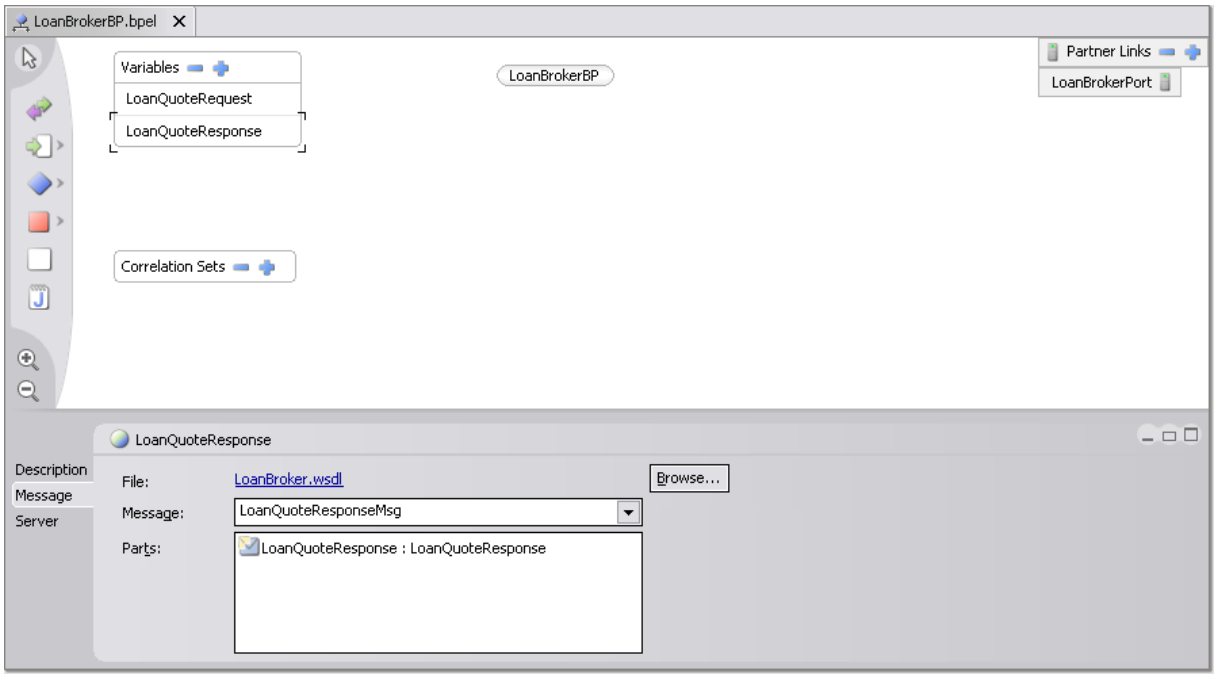

Figure 4.3.4: Declare request and response variables

Step 6 – Add a receive activity

Now the process is ready for adding a receive shape which will accept incoming requests on the port type "`LoanBrokerPort`". After adding the shape we have to connect it to the corresponding partner link select the operation and assign the incoming message variable. By now the process can receive an incoming "`LoanQuoteRequestMessage`" on the appropriate port and is able to work with the therein included values.

Step 7 – Interaction with the credit bureau

The credit banks need more information about the customer than the data which is contained in the loan broker request message. So the message has to be enriched with more data. Beside the SSN, the loan term and the loan amount, the bank also needs the customer's credit score and history. Because of that the message has to be augmented with data delivered from a credit bureau. At this point the pattern *'Content Enricher'* comes in place.

Add a new "*Sequence*" shape to the business process and call it '`Content Enricher'`. You don't need to add a sequence to the process but we're doing so to get a better overview later on when the process may increase in volume. The first thing to do is to add a new partner link, name it '`CreditBureauPort`' and import the '`CreditBureau.wsdl`' file into the partner link. After that you have to add two variables to the business process. The first is called '`CreditBureauRequest`', the second is called '`CreditBureauResponse`'. The message types of each of them is defined in the '`CreditBureau.wsdl`' file previously imported.

Construct credit bureau request

If we want to call the credit bureau web service it is necessary to create the request message. We will perform this operation by adding a '*Transform*' activity to the business process inside the content enricher sequence. Select the incoming

message (‘LoanQuoteRequest’) and the resulting message (‘CreditBureauRequest’) and create a new transformation service by selecting the new button in the transformation service mask. Fill out the fields with the desired values and press finish. The next window to appear is one in which you can select the input fields and map them to the wanted output fields (see Figure 4.3.5).

Invoke the credit bureau Web Service

After creating the credit bureau request message the process is ready to call the credit bureau Web Service. To do so walk through the following steps:

- Add an ‘*Invoke*’ activity to the content enricher sequence and name it ‘Credit Bureau’.
- Set the partner link to the ‘CreditBureauPort’.
- Select the ‘GetCreditScore’ operation.
- Set ‘CreditBureauRequest’ as request message.
- And set ‘CreditBureauResponse’ as response message.

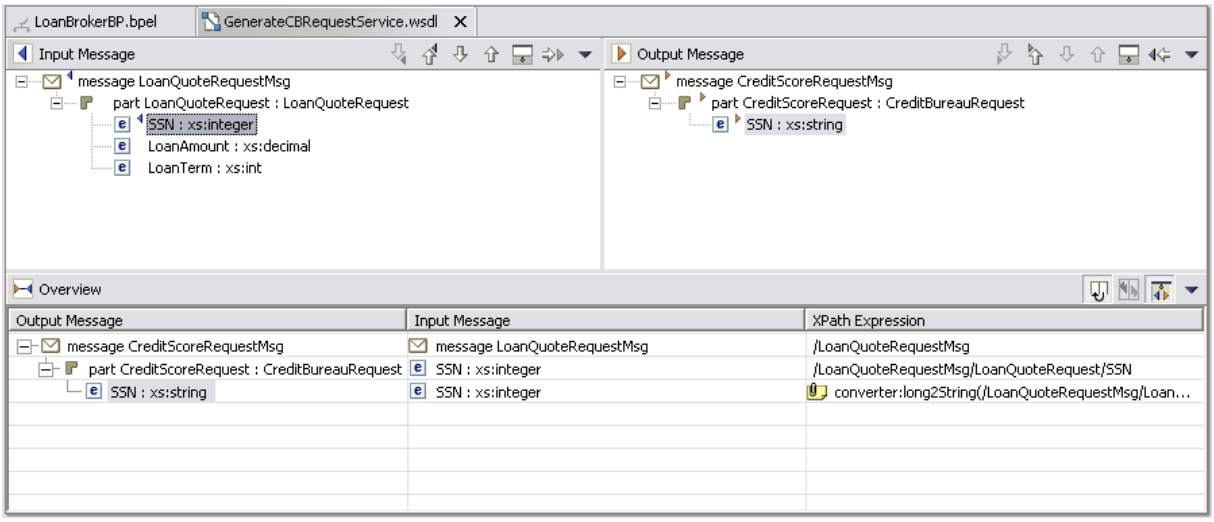

Figure 4.3.5: Create credit bureau request message

Enrich credit bank request

After the call to the credit bureau Web Service returns to the process we have to prepare the request message to the multiple credit banks. At first add a new variable called ‘BankQuoteRequest’ to the process. Set the type of the variable to the message type we defined at section 4.2 named ‘GetLoanQuoteRequest’.

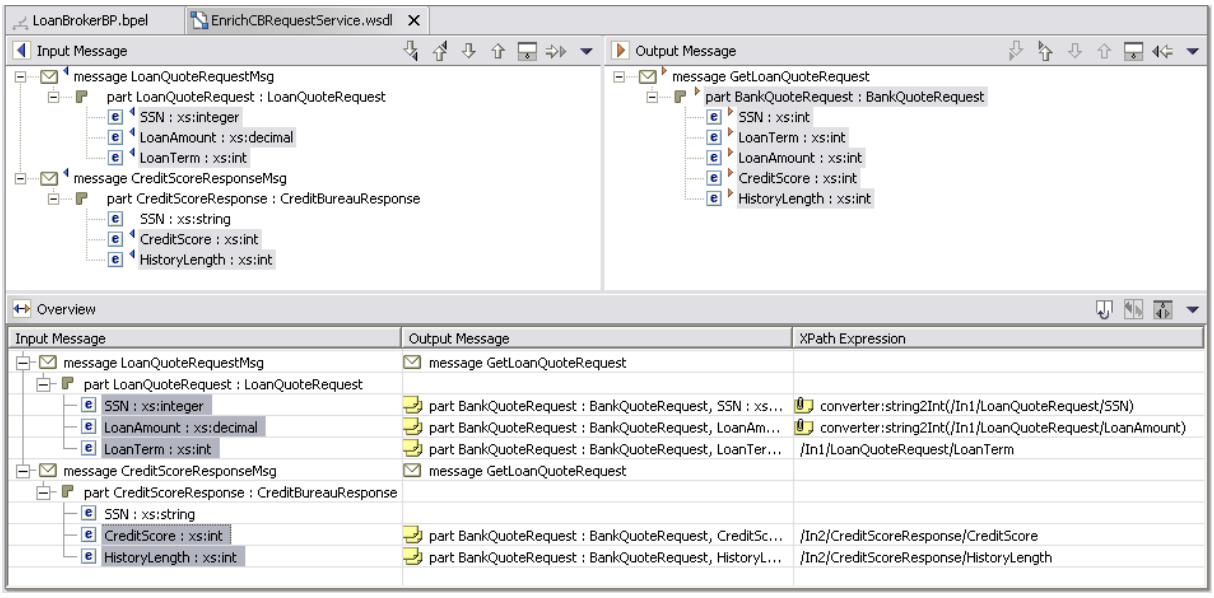

Figure 4.3.6: Enrich the credit bank request

The variable has to be filled with different values which originate from the '`LoanQuoteRequest`' as well as from the '`CreditBureauResponse`'. In Figure 4.3.6 the transformation is shown. You can see that we're using another Transformation activity at this point. The difference is that we get two input messages. In detail the mapping looks like this:

- LoanQuoteRequest.SSN → BankQuoteRequest.SSN
- LoanQuoteRequest.LoanAmount → BankQuoteRequest.LoanAmount
- LoanQuoteRequest.LoanTerm → BankQuoteRequest.LoanTerm
- CreditBureauResponse.CreditScore → BankQuoteRequest.CreditScore
- CreditBureauResponse.HistoryLength → BankQuoteRequest.HistoryLength

The output message is the message with which we can work at the multiple credit bank instance Web Services.

The whole '*Content Enricher*' with its activities is shown in Figure 4.3.7.

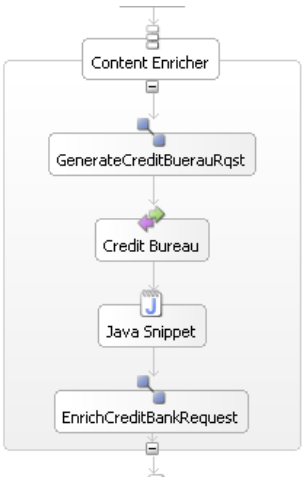

Figure 4.3.7: The content enricher within the loan broker process

Step 8 – Interaction with the banks

After enriching the request message with credit bureau information, the loan broker has the needed data to interact with the banks. As discussed in chapter 3 "Mapping Patterns to WSAD-IE" we have to implement the '*Recipient List*' to route a bank request to the according bank. Before a message will be sent to a credit bank the recipient list checks upon given preconditions which credit banks have to be contacted. The requirements for each bank were shown in section 2.2 "Designing with Patterns" and will be listed later on in this section. After ascertain from which credit banks to obtain a loan quote, the appropriate messages for each bank have to be constructed.

Insert the recipient list frame

The '*Recipient List*' is a flow in which probably all three credit bank Web Services are called in parallel. To implement that in WSAD-IE we use the '*Flow*' activity of BPEL. Within this flow we have to implement multiple switch activities followed by sequences.

Start first by inserting a '*Flow*' activity into the business process and rename it to '`Recipients List`'. Because we have to invoke up to three banks we need to copy the common '`BankQuoteRequest`' variable to the three individual request messages- each for every credit bank service. This is done by adding an '*Assign*' shape to the '*Flow*' activity. Afterwards we're putting two '*Switch*' activities into the flow and one '*Sequence*' activity and connect the 'assign activity with each of these activities. Inside the switch activities we add a case condition which represents the filter rule. Now we got our frame (see Figure 4.3.8) which will be filled up with more activities and implementation code later on in this step.

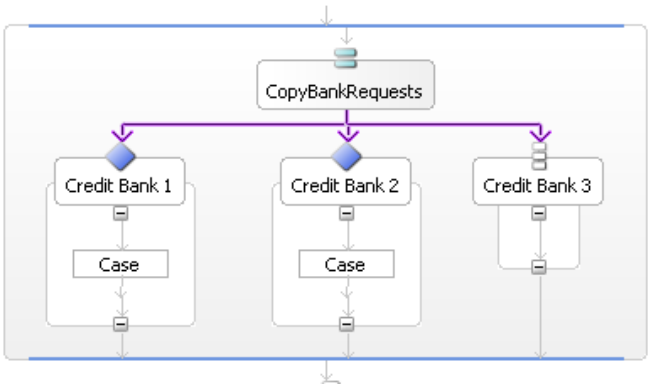

Figure 4.3.8: The recipient list frame

Implement credit bank 1 branch

Inside the credit bank 1 switch activity select the case statement. Within the properties window select 'condition' and 'java statement' as value. As java statement write the following condition:

```
boolean result = false;
BankQuoteRequest bankRqst = getBankQuoteRequest().getBankQuoteRequest();
if((bankRqst.getCreditScore() >= 500) &&
   (bankRqst.getHistoryLength() >= 5)) {
  result = true;
}
return result;
```

Listing 4.3.1: Credit bank 1 decision rule

Insert an '*Invoke*' activity after the case rectangle. This activity needs a partner link. Import the Web Service binding of the credit bank 1 (`CreditBank1JavaService.wsdl`) and set the accordant partner link of the invoke activity. Select '`GetLoanQuote`' as operation and insert new variables for the incoming and outgoing messages. Click on the new button beside the request field and enter '`Bank1LoanQuoteRequest`' as variable name. Repeat this step for response variable and name it '`Bank1LoanQuoteResponse`'. At this point the process is ready to interact with the first credit bank.

Implement credit bank 2 branch

The branch of the credit bank 2 looks very similar to the one before. The differences are the condition of the case statement (see Listing 4.3.2) and the partner link. As partner link of the '*Invoke*' activity the file `CreditBank2JavaService.wsdl` has to be imported. The needed variables should be named '`Bank2LoanQuoteRequest`' and '`Bank2LoanQuoteResponse`'.

```
boolean result = false;
BankQuoteRequest bankRqst = getBankQuoteRequest().getBankQuoteRequest();
if((bankRqst.getCreditScore() >= 700) &&
   (bankRqst.getHistoryLength() >= 10)) {
  result = true;
}
return result;
```

Listing 4.3.2: Credit bank 2 decision rule

Implement credit bank 3 branch

The last of the three credit banks does not have a '*Switch*' activity. That means that this branch will always be contacted when going through the recipients list. This is because the last bank will accept every loan quote request sent to it. So the only thing we have to do is to add an '*Invoke*' activity, set the partner link to the imported file '`CreditBank3JavaService.wsdl`' and create the corresponding variables called '`Bank3LoanQuoteRequest`' and '`Bank3LoanQuoteResponse`'.

Multiply the request messages

Last thing we have to go through is to create copies for each of the credit bank variables. In WSAD-IE this is done by the BPEL activity '*Assign*'. We already inserted this activity to our process. Now we have to implement its behavior. WSAD-IE offers the possibility to easily build the assign statements. As shown in Figure 4.3.9 the part '`BankQuoteRequest`' out of the common '`BankQuoteRequest`'-variable is copied to the part '`BankQuoteRequest`' of the '`Bank1LoanQuoteRequest`'-variable. After the execution of the assign activity

the request variable for the credit bank 1 (i.e. the Customer Bank) is filled with the needed values.

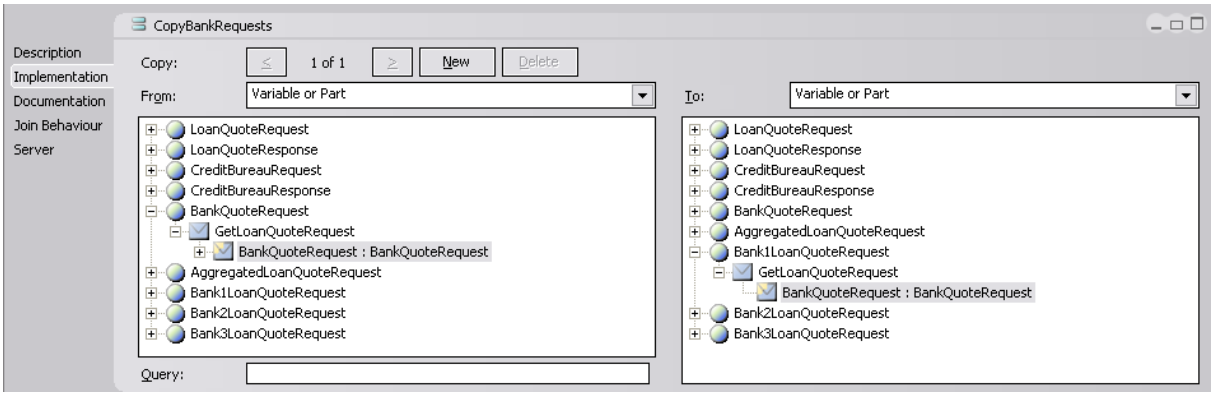

Figure 4.3.9: Copy the bank requests

Repeat this step for each of the credit bank requests. The '*Assign*' activity allows implementing more than one of this copying operations. Simple press the new button at the top and set the other assigns.

After completing this step the process is ready for interacting with different credit banks (see Figure 4.3.10). Which bank will be called is decided according to the result the credit bureau delivers at the point of the '*Content Enricher*' in the business process.

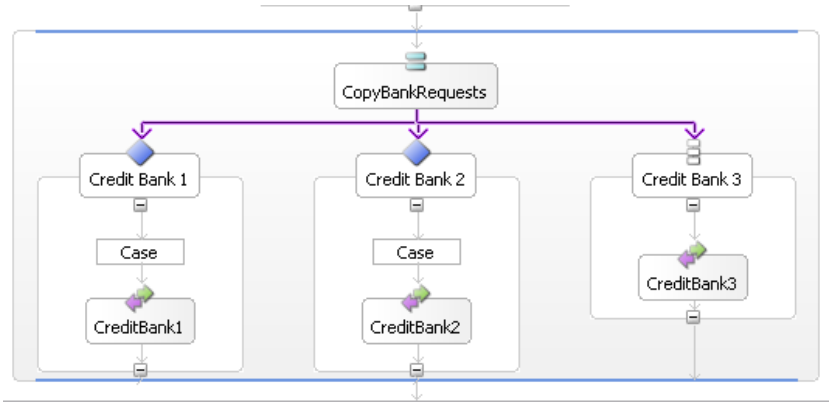

Figure 4.3.10: The whole recipients list

Step 9 – Build the aggregator

Last step in building the loan broker business process is to aggregate multiple messages delivered by multiple banks into one single message. This is not only done by adding each message but rather by an algorithm which determines which of the incoming loan quote requests is the best and returns this as best loan offer to the client, i.e. the loan broker business process. The aggregator will be integrated as Web Service. The implementation will be a Java class which offers two different methods – one for adding messages and one for selecting the best offer of the added messages.

Due to the Java implementation we have to pay attention on correlation. The service needs additional information about the caller of the method. The singleton instance of the aggregator service could not determine from which client the request originated if we would leave that information out. Therefore we

add a parameter called '`processID`' to the message signature beside the message parameter. The creation and addition of the correlation ID is the representation of the '*Correlation Identifier*' pattern.

The completeness criterion is very simple. The aggregator Web Service will be appended after the parallel interaction flow of the credit banks. So if the activity thereafter will be reached all possible bank responses already arrived and the aggregator can calculate the best offer.

To create the aggregator we first have to design the structures for adding a bank response message and for the best loan offer request message. The first structure was already designed a few steps earlier (see Section 4.2 "The Bank Web Services" and Figure 4.2.2). It is nothing more than the response message of the credit banks. The best loan offer message is shown in Figure 4.3.11.

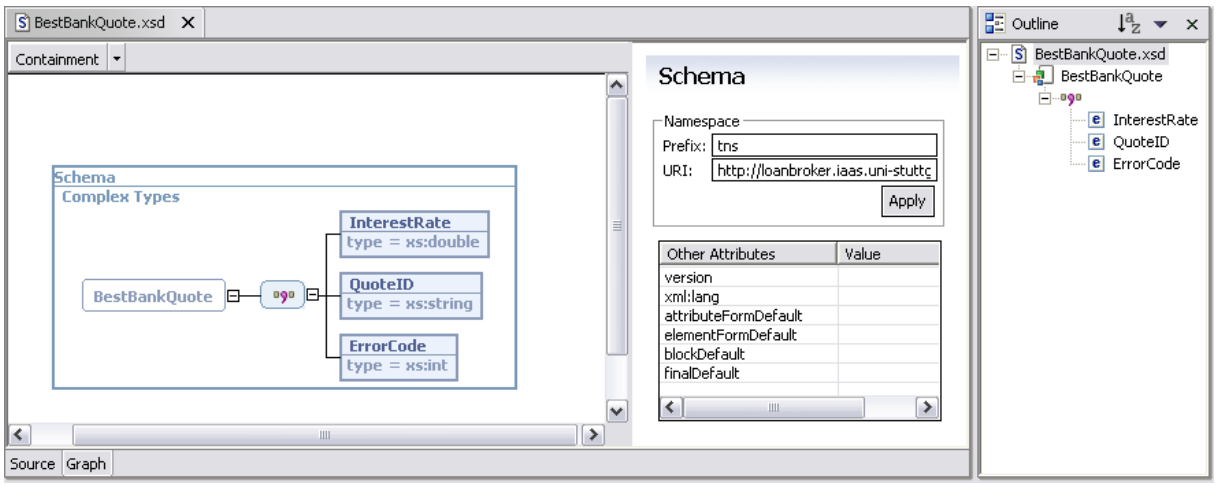

Figure 4.3.11: The best loan offer message structure

The Java implementation of the aggregator Web Service uses a Java representation of the XSD Schemas of the two message structures as parameters of the methods. The first Java representation was generated during the creation of the credit bank Web Services. The second representation has to be generated with the generator included in WSAD-IE. Afterwards a Java file named '`BestBankQuote.java`' exists.

The next step is implementing the '`Aggregator.java`' class. Because the service has to aggregate more than one message of different processes, the service has to implement a mechanism so that it can correlate the incoming messages with a process instance. We won't handle that with the help of BPEL correlation set facility. Rather the implementation itself has to care about the correlation of different messages of different clients (see Listing 4.3.3). Therefore the process generates a unique id and adds it to the message dedicated for the aggregator service.

```
public class AggregatorImpl {
   private static AggregatorImpl _instance = null;

   private AggregatorImpl () {
   }

   synchronized public static AggregatorImpl getInstance() {
```

```
      if (_instance == null) {
        _instance = new AggregatorImpl();
      }
      return _instance;
    }

    private Hashtable allProcesses = new Hashtable(5);

    synchronized public void addBankQuoteResponse(
      BankQuoteResponse bankQuoteResponse, int processID) {
      String id = (new Integer(processID)).toString();
      Object processRepliesObj = allProcesses.get(id);
      Hashtable processReplies;
      if (processRepliesObj == null) {
        processReplies = new Hashtable(5);
        allProcesses.put(id, processReplies);
      } else {
        processReplies = (Hashtable) processRepliesObj;
      }
      processReplies.put(bankQuoteResponse.getQuoteID(), bankQuoteResponse);
    }

    synchronized public BestBankQuote getBestBankQuote(int processID)
    {
      String id = (new Integer(processID)).toString();
      BestBankQuote result = null;
      double interestRate = -1;
      String quoteID = null;
      Hashtable process = (Hashtable) allProcesses.get(id);
      if (process != null) {
        Enumeration elems = process.elements();
        while (elems.hasMoreElements()) {
          BankQuoteResponse element =
            (BankQuoteResponse) elems.nextElement();
          if ((quoteID != null) &&
              (element.getErrorCode().intValue() == 0)) {
            if (interestRate > element.getInterestRate()) {
              interestRate = element.getInterestRate();
              quoteID = element.getQuoteID();
            }
          } else {
            interestRate = element.getInterestRate();
            quoteID = element.getQuoteID();
          }
        }
      }
      if (quoteID != null) {
        BankQuoteResponse best = (BankQuoteResponse) process.get(quoteID);
        BestBankQuote resultBBQ = new BestBankQuote();
        resultBBQ.setErrorCode(best.getErrorCode().intValue());
        resultBBQ.setInterestRate(best.getInterestRate());
        resultBBQ.setQuoteID(best.getQuoteID());
        result = resultBBQ;
      }
      return result;
    }
}
```

Listing 4.3.3: Implementation of the Aggregator Servie (AggregatorImpl.java)

Due to the Web Service implementation a singleton class can not be wrapped as a service because the needed public constructor isn't available. That's the reason

why another class- we call it the port type class- has to be implemented (see Listing 4.3.4). This class uses the singleton pattern inside and can be wrapped as a Web Service.

```
public class AggregatorPT {

   private AggregatorImpl _instance = null;

   public AggregatorPT() {
      if (_instance == null) {
        _instance = AggregatorImpl.getInstance();
      }
   }

   synchronized public void addBankQuoteResponse(
        BankQuoteResponse bankQuoteResponse, int processID) {
      _instance.addBankQuoteResponse(bankQuoteResponse, processID);
   }

   synchronized public BestBankQuote getBestBankQuote(int processID) {
      return _instance.getBestBankQuote(processID);
   }
}
```

Listing 4.3.4: The Aggregator Port Type class (AggregatorPT.java)

After implementing the aggregator class, wrap it inside a Web Service. Use the '*Service build from Java*' tool within WSAD-IE to generate the required WSDL files. After processing we can import the newly generated Web Service description into our loan broker process. Now we're ready for building the '*Aggregator*'.

At first insert a '*Flow*' activity after the '*Recipients List*' flow and rename it to '`Aggregator`'. The next step is to insert three '*Switch*' activities each containing one '*Case*' element. This construction let us decide which of the banks did answer to the request. Before we can add an '*Invoke*' activity to each branch we have to implement the case elements. These queries will simple check if the bank responses are null (i.e. the bank didn't respond) or not (e.g. see Figure 4.3.12).

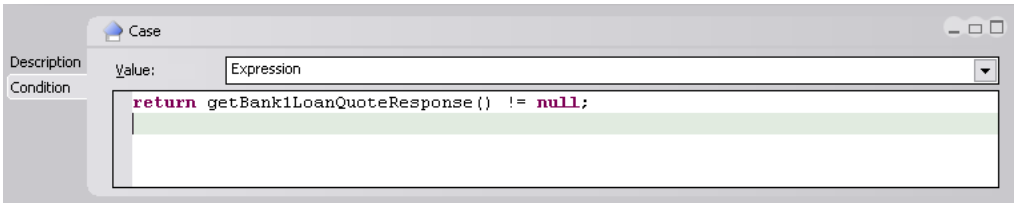

Figure 4.3.12: Check if credit bank responses are null

Now add '*Invoke*' activities to each branch and set the partner link to the imported "Aggregator Web Service". As operation select the '`AddBankQuoteReply`' method. The input variable is '`Bank1LoanQuoteResponse`' for the credit bank 1, '`Bank2LoanQuoteResponse`' for credit bank 2 and '`Bank3LoanQuoteResponse`' for the last credit bank. Because of the generated WSDL file of the bank Web Services we have also to specify the response variables although the responses will always be null. That's

because WSAD-IE isn't able to create a one-way-invoke out of a void return value of the Java method signature.

To finish the aggregator insert another '*Invoke*' activity to the '`Aggregator`' flow. Set the partner link of this activity to the earlier imported aggregator Web Service description. Select '`GetBestBankQuote`' as operation within the implementation section and set both the request and response message variables though the input parameter of the implemented method doesn't exist. The important variable is the response variable because it's including the best bank quote of the previously aggregated bank offers. The last step is to connect all three branches where the individual messages are aggregated with the lastly added '*Invoke*' activity. In Figure 4.3.13 you can see the whole Aggregator part of the Loan Broker business process.

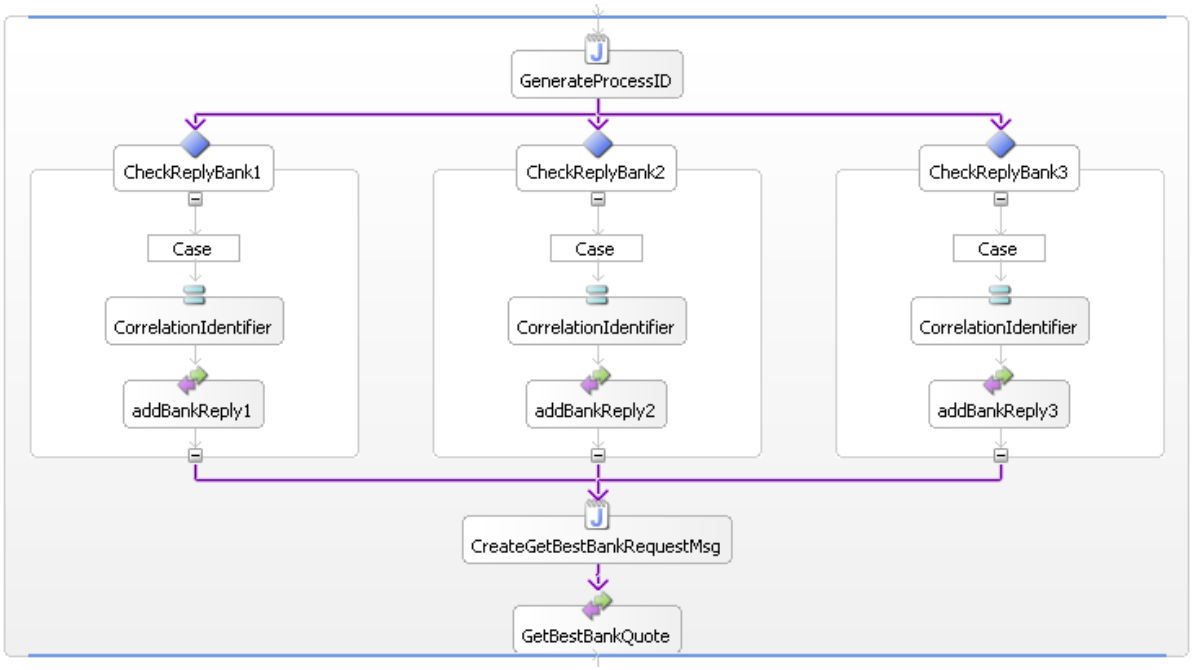

Figure 4.3.13: Aggregator part of the loan broker business process

Now the process is ready for constructing the reply message for the client. After this following step the implementation of the process is finished.

Step 10 - Construct the reply message

Last action to be performed is to construct the reply message. In the patterns scenario this is the part of the message translator. To achieve the translation we need a so called '*Transform*' activity. To form the output message we need data of the input request message and the response of the last called Web Service. Consequential the '*Transform*' activity needs the input messages '`LoanQuoteRequest`' and '`BestBankQuote`' and responds with the filled '`LoanQuoteResponse`' message as output message (see Figure 4.3.14).

Step 11 – Add a reply activity

The last step in implementing the business process is to add the '*Reply*' activity at the end of the process. After adding the activity set the partner link to the WSDL file which we imported in Step 5. As operation select '`GetLoanQuote`' and

as message select '`LoanQuoteRequest`'. Now the process is ready and can be run after the deployment following in the next steps.

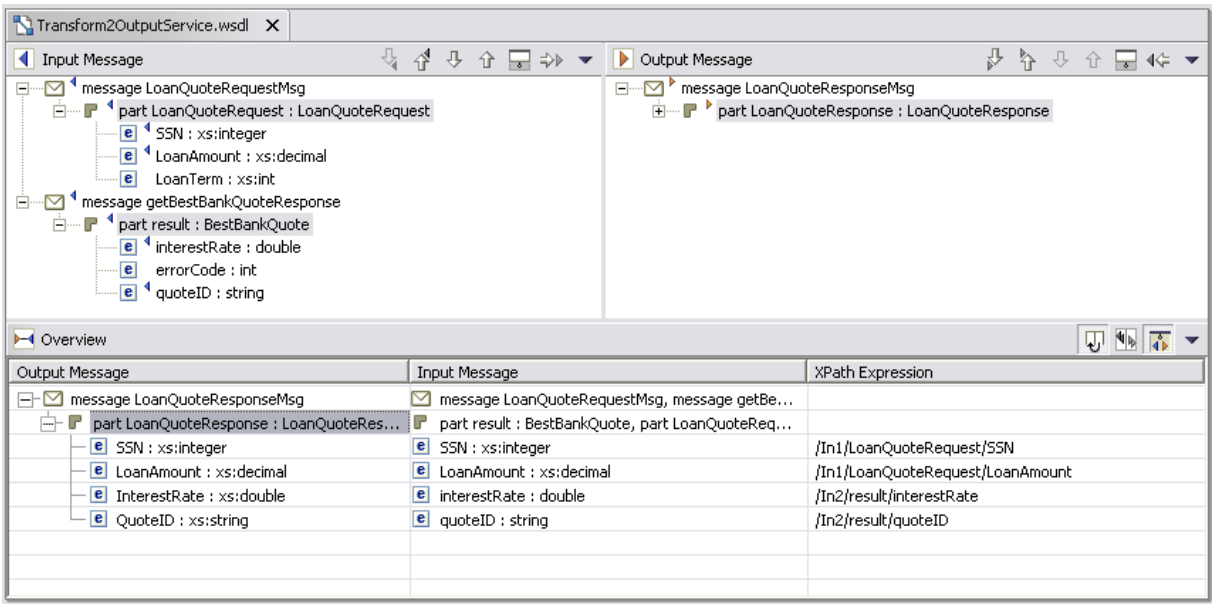

Figure 4.3.14: The transformer for the reply message

Step 12 – Deploy the process

After creating the business process it should be deployed on an application server. WSAD-IE provides us with functionality of generating deployment code. Because we only want to have the ability to call the process from the outside over SOAP/HTTP we do not need to adjust anything. We only have to check the partner links, if they are pointing to the right service implementations and start the generation.

Step 13 – Publish the generated application

After generation of the deployment code of the loan broker process publish the application containing the process on the WebSphere Application Server. First of all add the previously generated EAR project to the server's application projects. Afterwards publish the application. This is done by selecting the option in the context menu entry of the server. The job is done automatically and the database is being updated so that no further interaction is needed. Keep in mind that the loan broker process uses the credit bureau process to enrich the incoming messages. Because of that the EAR containing the credit bureau business process has to be published on the server as well.

Step 14 – Start the application

After publishing the application it's required to start the application in order to use it. This is done by selecting the appropriate menu item out of the context menu entry of the server. The Server tries to start the application. The result is shown on the server's console.

Step 15 – Test the application

After publishing and starting the application on the server we can test the business process with the help of WSAD-IE. The integrated "*Business Process*

Web Client" offers the possibility among other things to test business processes. This is simply done by entering the values of the request message into a web front-end. Afterwards the web client shows the results, i.e. the values of the response message, generated by the business process. The following two screenshots show this proceeding.

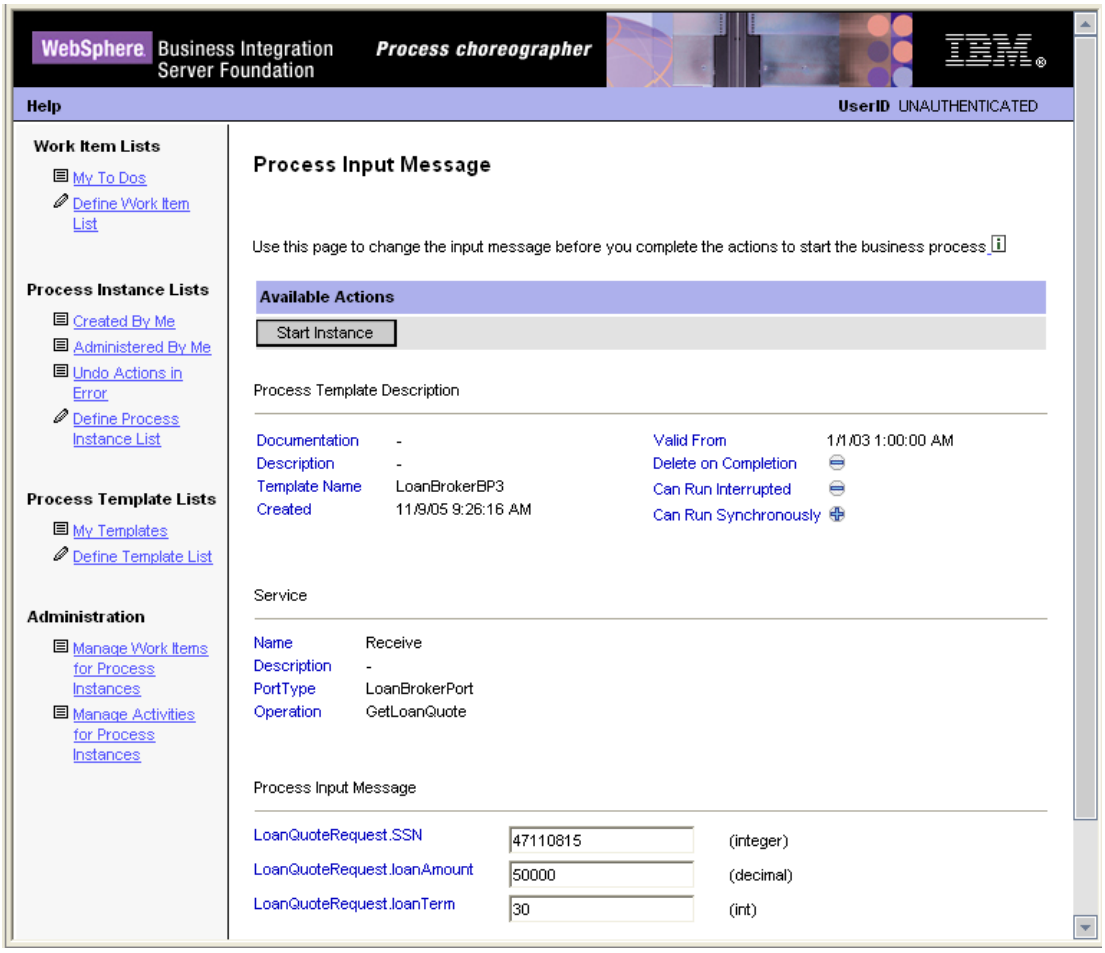

Figure 4.3.15: Enter request values for a business process

As shown in Figure 4.3.15 following input values were entered for testing:

- SSN: 555888444
- Loan amount: 50000
- Loan term: 30

The process answered with the values shown in Figure 4.3.16. During the computation of the best loan offer the process contacted the credit bureau to enrich the request dedicated for the banks. In our case the credit bureau returns the following values as result of the random calculation:

- Credit score: 557
- History length: 7

Because of these values the process contacted only two banks within the '*Recipient List*' – the *Consumer Bank* and the *Loan Shark*. As expected the best loan quote is offered by the Consumer Bank.

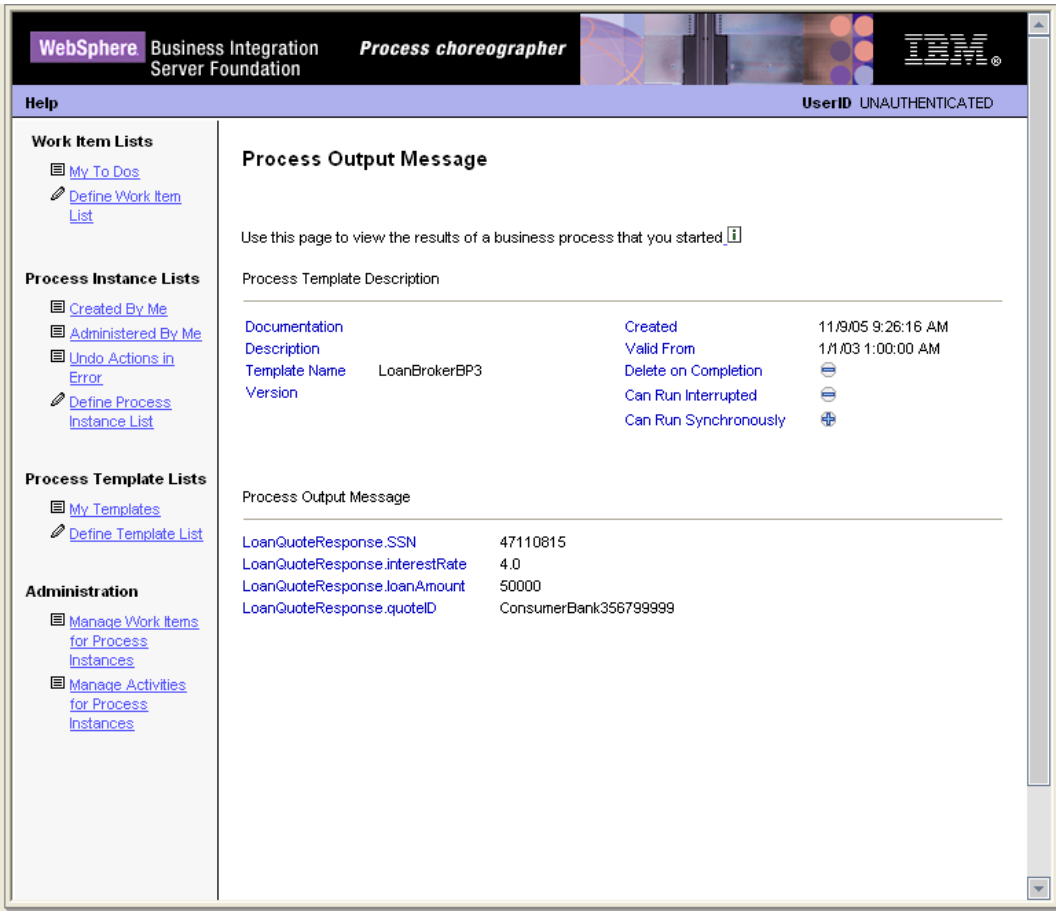

Figure 4.3.16: Overview of the result of the loan broker business process

5. Conclusions

In the Technical Report we have seen how a sample scenario can be modeled with the help of enterprise integration patterns. Afterwards we were discussing how you can map these patterns onto a BPEL compliant developing tool. Within this discussion we have seen how single BPEL elements can be combined to build an enterprise integration pattern. The last step included the deployment and execution of the whole business process.

In some place we might implement the patterns a different way. E.g. we could have built the recipient list as sequential calls to the banks Web Services. Our solution is just a proposition how to solve the problem scenario. Another thing is that some parts of the overall business process engage some IBM specific BPEL features. If you would want to build a general solution you have to substitute these elements by BPEL4WS compliant parts.

Our example doesn't cover the whole spectra of the rich set of enterprise integration patterns. We just present a little extract of the powerful pattern catalogue. Furthermore we disregard some very critical issues in a common scenario. We didn't look at error handling procedures e.g. in absence of services, malformed messages or unanswered requests. But all we talked about is much enough to demonstrate how enterprise integration patterns can be used and mapped onto an executable environment.

The whole purpose of the technical report was in showing how you can design a common scenario and divide it into several parts with the help of integration

patterns. The main aspect was to show how a given architecture can be mapped onto an executable runtime language like BPEL with the help of a BPEL runtime engine and modeling tool like WSAD-IE.

6. References

1. G. Hohpe, H.-S. Tham; Enterprise Integration Patterns with BizTalk Server 2004; ThoughtWorks Whitepaper; July 2004; http://www.eaipatterns.com/docs/integrationpatterns_biztalk.pdf
2. G. Hohpe, B. Woolf; Enterprise Integration Patterns; Addison Wesley; 2004
3. M. Kloppmann, D. König, F. Leymann, G. Pfau, D. Roller; Business process choreography in WebSphere: Combining the power of BPEL and J2EE; IBM Systems Journal, Volume 43, Number 2, 2004
4. M. Kloppmann, G. Pfau; WebSphere Application Server Enterprise Process Choreographer – Concepts and Architecture; IBM Corporation (December 2002); http://www-128.ibm.com/developerworks/websphere/library/techarticles/wasid/WPC_Concepts/WPC_Concepts.html
5. T. Andrews, et al; Business Process Execution Language for Web Services (BPEL4WS) Version 1.1 (May 2003); ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf
6. S. Weerawarana, F. Cubera, F. Leymann, T. Storey, D.F. Ferguson; Web Services Plarform Architecture; Prentice Hall; March 2005
7. E. Gamma, R. Helm, R. Johnson, J. Vlissides; Design Patterns: Elements of Reusable Object-Oriented Software; Addison Wesley; 1997